Transforming monolithic architecture towards microservice architecture

Miika Kalske

Helsinki November 19, 2017

M.Sc. Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Miika Kalske

Työn nimi — Arbetets titel — Title

Transforming monolithic architecture towards microservice architecture

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M.Sc. Thesis	November 19, 2017	72 pages

Tiivistelmä — Referat — Abstract

Monolithic architecture has been the standard way to architect applications for years. Monolithic applications use a single codebase which makes the deploying and development easier without adding any additional complexity as long as the size of the application stays relatively small. When the size of the codebase grows the architecture might deteriorate. This slows down the development and making it harder to on-board new developers. Microservice architecture is a novel architecture style that tries to solve these issues in larger codebases. Microservice architecture consists of multiple small autonomous services that are deployed and developed separately. Microservice architecture enables more fine-grained scaling and makes it possible to have faster development cycles by decreasing the amount of regression testing that is needed, because each of the services can be deployed and updated separately from each other.

Microservice architecture provides also multiple new challenges that have to be solved in order to get the benefit from them. These challenges are such as the handling of distributed transactions, communication between microservices, separation of concerns in microservices and so on. On top of the technical challenges there are also organizational and operational challenges. The operational challenges are such as monitoring, logging and automated deployment of microservices.

This thesis studies the differences between monolithic and microservice architecture and pinpoints the main challenges on the transition from monolithic architecture to microservice architecture. A proof of concept on how to transform a single bounded context from monolith to microservices will be made to get a better understanding of the challenges. Also a plan how to migrate tangled bounded contexts from monolith to microservices will be made in order to fully support the transition process in the future. The results from the proof of concept and the plan that was made show that the cohesion and loose coupling is more likely to stay when the bounded context is transformed to microservice.

Avainsanat — Nyckelord — Keywords
microservices, architecture, refactoring

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction **1**

2 Background and motivation **3**

2.1 Motivation 3

2.2 Monolithic architecture 5

2.3 Microservice architecture 9

2.4 Main differences of monoliths and microservices 14

2.4.1 Development 15

2.4.2 Database 16

2.4.3 Scaling 17

2.4.4 Production environment 18

2.4.5 Testing 19

2.4.6 Continuous integration and continuous deployment 19

2.4.7 Summary 22

3 Challenges when transforming from monolith to microservices **22**

3.1 Database challenges 22

3.2 Microservice premium 26

3.3 Communicational challenges 26

3.4 Testing challenges 27

3.5 Observability challenges 28

3.6 Organizational challenges 29

4 Techniques for transforming monolithic to microservices architecture **30**

4.1 Enablers 31

4.1.1 Testing 31

4.1.2 Organization 33

4.2 Techniques 34

5 Transforming an existing application towards microservices 40

5.1 Case introduction 41

5.2 Techniques used in proof of concept 43

5.3 Proof of concept 46

5.3.1 Attachment microservice proof of concept 47

5.3.2 Transformation plan for invoices 51

5.4 Results of proof of concept 58

5.5 Future Suggestions 60

6 Conclusions 63

References 65

1 Introduction

Microservice architecture is a new architectural style that has emerged in the last few years. There is no precise definition of microservice architecture but to summarize microservices are a collection of loosely coupled small autonomous services [LF14]. These services focus on doing one thing well and thus adhering to single responsibility principle. In monolithic architecture, everything is developed and deployed as a single artifact. This makes the initial development easy and simple to understand. However, as the codebase grows in size, also the problems in monolithic architecture begin to show. The main problems are the large codebase which makes development slower, the difficulty of continuous deployment and limited scaling possibilities.

Microservices have many strengths compared to monoliths. Microservices put the service boundaries where the business boundaries are thus making it visible where the functionality is located in the codebase [New15a]. This is extremely useful in large codebases because their monolithic counterparts it can be hard to specify the place where certain functionality is located. Microservices make it possible to use different technologies as different services can be implemented with various technologies. This means that the adoption of new technologies can be faster with microservices [New15a]. Microservices also enable continuous deployment and decentralization of data. Thus microservices set up the grounds for more rapid innovation and the possibility for the companies using microservices to gain competitive advantage.

Microservices cannot be considered as a silver bullet that solves every problem. Instead, they come with their challenges also. Creating a distributed system is a complex problem. Different services have to communicate with each other, using traditional transactions is not possible with microservices, testing can be more difficult, running microservices in production poses its problems and on top of the technical challenges, there are also organizational challenges.

The decision whether to go with microservices or monolithic architecture comes down to which challenges are easier to solve in the long run. Typically organizations have moved towards microservices when their existing monolithic codebase has become too complex, hard to scale and the system is not resilient enough. From these reasons we can conclude that the microservice architecture is more suitable for existing organizations that have grown big enough and have a good understanding about their business landscape [LF14].

This thesis provides a real-world proof of concept about the transformation from existing monolithic application towards microservices. The proof of concept will be done for Procountor software that is a financial management software that has been under development for over 15 years. As the existing work considering microservices is limited in the sense that there are not many academic publications about the subject and the number of cases studies is lacking this thesis provides more insight to this problem.

The result of this thesis is a suggestion on how the architecture of Procountor software should evolve in the future. The proof of concept together with the plan on how to transform tangled business contexts to microservices is expected to give a good enough guess about the difficulty of the transition. Both of these should highlight the main challenges of the transition in the scope of building a single microservice and also in the broader scope of the whole architecture transformation. As the transition period is very long and the transition can take multiple years it makes sense to divide the transformation to smaller subsets.

The thesis is structured as follows. Chapter 2 gives the background and motivation for the transformation of monolithic architecture towards microservices architecture. Both architectural styles will be explained in length, and their strengths and weaknesses will be discussed together with the main differences between these two architectural styles.

Chapter 3 goes in to details about the challenges that the transition to microservice architecture provides. These challenges can be divided to technical and organizational challenges. The main focus will be on the technical challenges such as how to handle transactions, splitting up bounded contexts, how to handle communication between different microservices and many more.

Chapter 4 describes the techniques which can be used when transforming a monolithic application to microservices. Also, the enablers that are required before the transition are listed.

Chapter 5 introduces the case study of this thesis. A proof of concept will be made together with a plan how to transition from monolithic architecture to microservice architecture in Procountor. The proof of concept consists of transitioning one business context to microservices. The plan gives a detailed view on how to transition legacy code to microservices. This plan can also be used in other organizations which are considering the transformation towards microservices architecture. At the end of the chapter results of the proof of concept together with the future suggestions about the architecture will be presented.

2 Background and motivation

This chapter contains the background and motivation of the transformation from monolithic architecture towards microservices. Monolithic architecture and microservice architecture are both described in this chapter. These two architectural styles are compared to each other, and use cases where they are applicable are presented. This information is valuable later on when the actual proof of concept considering transformation from monolith to microservices is discussed in Chapter 4. The main point of this section is to give the tools to understand the challenges that are being solved and why this transformation is being done.

2.1 Motivation

Microservice architecture is a novel distributed software architecture that provides multiple benefits in the cases of large applications with big monolithic codebase. Even though microservice architecture has its challenges, it can still be considered a better solution in situations where the size and complexity of the codebase have reached certain limits [Fow15a]. The biggest advantages that microservice architecture provide for large applications are better scaling of the application and faster development in large scale applications. Faster development cycles are possible, because of the small services which make it easier to use continuous integration (CI) and continuous deployment (CD) [BHJ16].

Rapid innovation is especially important in the current software landscape where the paradigm has shifted from desktop PCs and corporate server rooms to cloud [Hay08]. This change enables easier updates even with monolithic applications, but microservice architecture together with CD enables organizations to really take advantage of the cloud computing and the possibility for frequent updates and scaling that it provides.

CI requires developers to integrate their code with others' code regularly [BHJ16]. A CI pipeline pulls the code from version control regularly and runs the automated tests on the code. Tools such as Jenkins can be used to build the CI pipelines quickly. Using CI with a monolithic application is advised as its usage avoids the possible integration problems. However using CI with a monolith can be hard when the codebase grows large enough because of longer build times, longer test runs and multiple people checking in code at the same time. With microservice architecture, there are numerous different services that have small codebases, which means that

the number of CI pipelines is higher. Even though the number of pipelines is higher the pipelines are a lot faster, which leads to more rapid feedback cycles. Because of the large number of services, CI is a necessity for microservice architecture [BHJ16]. CI is the first step towards CD.

CD is built on top of the CI pipelines. CD means that the application is deployed automatically to production. CD enables continuous user feedback and the possibility to learn from real user data to tune the application better for the users [OAB12]. CD is very tough to achieve with monolithic architecture because the changes can affect multiple places in the codebase. This means that verifying whether a build was successful takes a lot of time because of the time that it takes to run a fully automated test suite [Nai16]. Because of the long verification cycle teams with monolithic architecture typically stay away from the CD approach [OAB12].

Microservices makes CD possible by having small and separate services which can be verified faster. This enables fast release cycles together with possibilities to learn and to respond faster to the needs of the users. Deployments to production can happen even on a daily schedule. Organizations using microservices and CD can now adapt faster to the changing needs of the users and even use instant user feedback as a way to experiment and test what the user needs [OAB12].

The fast development together with CI and CD pipelines gives an edge in the competition for organizations that are using microservices and CD [OAB12]. Those organizations are truly agile and can respond to the needs of the users promptly. With the fast feedback cycle, organizations can experiment faster and find out early what features are going to solve the problems of users and what features are not required.

Microservice architecture provides better scaling than monolithic architecture [New15a]. The nature of microservices enables scaling of a single service. Services that require the most resources can be scaled upwards by duplicating the service to multiple nodes. Services requiring fewer resources can be down-scaled. With monolithic architecture, the whole application always has to be deployed which makes scaling more coarse-grained. Together with the current development of cloud services that enable easy down and up scaling using microservices can lead to cost-savings.

Microservice architecture provides multiple good qualities of software architecture such as loose coupling, high cohesion, single responsibility principle and modularity. Because of these aspects and the small size of a single service, developing new features can be fast even in a larger application [New15a].

Because of all these good qualities microservice architecture is a good choice for applications utilizing monolithic architecture that struggle with scaling or slow development cycle. Microservice architecture provides answers for these challenges but it should not be considered as a silver bullet because it has its own challenges, which will be presented later on this thesis.

2.2 Monolithic architecture

A monolithic application is an application that uses a single codebase to serve numerous different services and different interfaces such as REST (Representational state transfer), APIs and HTML pages [VGC+15]. The monolithic approach is considered as the standard way to start developing applications. A single codebase eases the development, deployment and scaling of the application as long as the size of the codebase is relatively small. A monolithic codebase is a good choice at the start of the project because of the aforementioned qualities and because there is not any distribution of code which would add complexity.

A monolithic application usually uses one single database to handle all the data. The database can be scaled to different partitions by sharding, but still, the partitions use the same schema. With a single database, transactions are usually easy to handle, as most database systems provide ACID (Atomicity, Consistency, Isolation, Durability) transactions. Developers can easily define transactions and focus more on providing new features to the end users. The single database has its limitations. The monolithic application might have multiple different kinds of data. Some of the data could be more suitable to be stored in a NoSQL database and some of the data in a relational database. However, with monolithic approach developers have to usually choose just one database engine and use that for all kinds of data.

Most applications can be pretty simple in the beginning, but as the application grows, so does its complexity. A typical way to handle complexity in an application that has a monolithic architecture is to split the application into different layers [Fow02]. A layered approach is widely used in networking and operating systems. Layered monolithic architecture that is displayed in Figure 1 on page 6 is very well known amongst developers, and everybody is familiar with this kind of architectural approach. The application is split into a UI layer, a service layer and a data access layer. Data access layer then usually accesses one database that handles all the data that is related to this application.

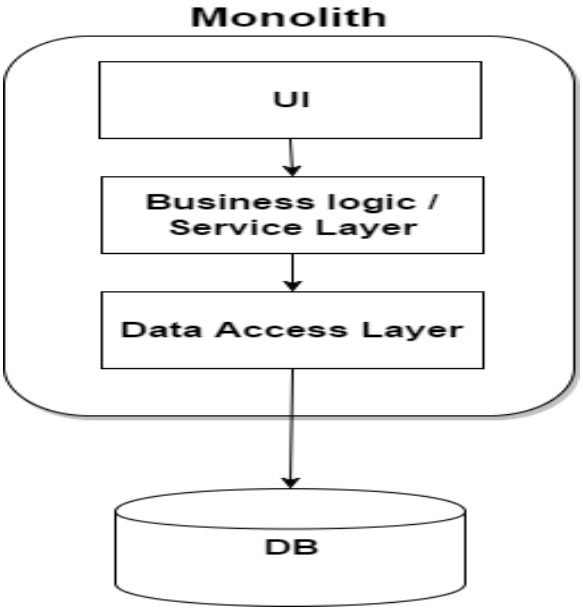

Figure 1: A typical monolithic application with n-tier architecture.

This kind of monolithic approach makes the development, deployment and scaling easy when the size of the application is moderately small [Ric15e]. Development is easy because the architecture is well known amongst developers and they know their way around the codebase.

Deploying a single artifact, for example in a Java application a single WAR-file, into testing or production environment is easy. This means that automating the deployment is simple. With a monolithic codebase, the deployment always contains every part of the application. The result is that when only one component is changed, the whole application has to be re-deployed. This makes continuous deployment hard [Ric15e]. Because every part of the application is always re-deployed organizations usually tend to have longer cycles between releases, which makes iterative development slower [Ric15e].

Scaling a monolithic application is done by adding new nodes with the same artifact. While this makes scaling very simple, it also decreases the scaling options. The components of the application that need more resources have to be scaled together with the components that could fill their workload with lesser resources. This means additional costs for the organization because the nodes require more resources the bigger the application is.

Easy development, deployment and scaling, while the codebase is small, makes monolithic architecture the best way to start developing a new application [Fow15b].

New features are easy to add, which means that the time to market is very fast and there are numerous existing frameworks that support the layered architecture very well. One such example is Spring Framework for Java applications [Piv17c].

However, as the size of the application and organization grows, so do the different layers of the application. Unless a lot of attention is paid to the architecture and quality of the codebase, it is very likely that the quality of the different layers deteriorates. The deterioration happens because of business requirements that force developers to make solutions which are not optimal. These sub-optimal solutions should be refactored but the time for refactoring can be hard to get, which leads to short-term solutions becoming long-term solutions. As the size of the codebase grows and the quality of the codebase deteriorates, it becomes harder to add new features and modify old features because the developer has to find the correct place to apply these changes [Ric15e]. This results in slower development cycles.

The development cycle of a new feature can slow down even more as the changes can affect multiple places. The impact of a change can be hard to understand [Kha15]. A developer might think that the change is small, but in reality it can affect multiple places. This leads to a situation where extensive manual testing is required and regression test cycles can thus become long [Kha15]. All this adds up and makes the process of releasing new features slow.

One of the downsides of a large monolith application is that it takes a long time to become familiar with the big codebase [Ric15e]. It takes time for new developers to get up to speed as they feel lost in the big codebase and cannot find the correct place to apply the changes. This can be avoided by keeping the modularity inside layers and continuously refactoring the codebase to keep the code clean. Good test coverage helps with refactoring [Fea04]. However, if the test coverage is lacking, developers might be afraid to refactor the codebase as their changes can affect multiple places and manually testing all these places is a big task [Fea04].

Clear modular boundaries inside a monolithic codebase can be hard to achieve and maintain during the development. Programming languages gives some tools for developers to ensure the modularity and loose-coupling in the monolithic codebase. For example, currently in Java it is possible to ensure some kind of boundaries by using the packages and visibility of the classes and methods. However, breaking these boundaries is easy because developers can change the visibility with ease and thus break the modularity. Because there are no hard modular boundaries in a monolithic codebase, it is possible that the modularity of the codebase decreases.

Also the quality of the code can decrease over time as it can be difficult for developers to understand how to correctly implement changes to the codebase [Ric15e].

This decreasing code quality makes it harder to write comprehensive tests. It is a vicious cycle which leads to a legacy codebase unless the developers and organization understand that continuous refactoring has to be made in order to keep the codebase clean [Fea04]. However, with disciplined developers and capable software architects, it is possible to have a monolith application with a good modular structure and good test coverage.

A large monolithic application with good test coverage can run into problems with the CI pipeline. The build times of CI pipelines can become longer as there is a lot of code to compile and thousands of automated tests to be run. When there are multiple people checking code in to a monolithic codebase it can result in a situation where the broken builds are someone else's problem and it becomes harder to pinpoint the problematic commit that broke the build [Nai16].

Even with a fast and reliable CI pipeline doing CD with a monolithic application is very hard and seldom done [Nai16]. With a monolithic codebase changes can have an effect on multiple places which means that regression testing has to be excessive. Releases have to be coordinated with multiple teams that can be working in different geographical areas.

These issues can lead to a situation where the organizations are too scared to deploy their applications continuously. Even though there are big challenges with using CD with monolithic codebase, there are examples of companies making it possible such as Etsy [Sch14]. CD with monolith requires a lot of specific tooling, using feature flags and care from the developers so that the deployment branch is always releasable.

Experimenting with new technologies and making changes to current technology stack is hard with a monolithic codebase [Ric15e]. The existing technology stack restricts the technology choices which can be made. Trying out new programming languages is limited in a monolithic codebase. For example, if the runtime environment is JVM, the programming languages that can be used are limited to ones that run on JVM [Ric15e]. Every technology choice that is made has to be thoroughly considered because as soon as a new technology has been introduced and adopted to the codebase, changing it to another can be extremely hard. Technology choices that are made have to be valid for years to come because big rewrites are expensive.

2.3 Microservice architecture

Microservice architecture is a novel architecture style, which has gained popularity in the last few years [LF14]. Microservices are small services that focus on one business context [New15a]. A rule of thumb about the size of a microservice could be that it can be rewritten in two weeks [New15a]. For example, one microservice could handle the creation of orders and another microservice could then handle the creation of an invoice that relates to the order. Figure 2 illustrates this example where the frontend of the application calls two different services, an order service and an invoice service. These services then separately handle the creation of the order and the creation of the invoice. These two services have their separate codebases and if there is a need to communicate between them, the communication is done through the APIs these services provide.

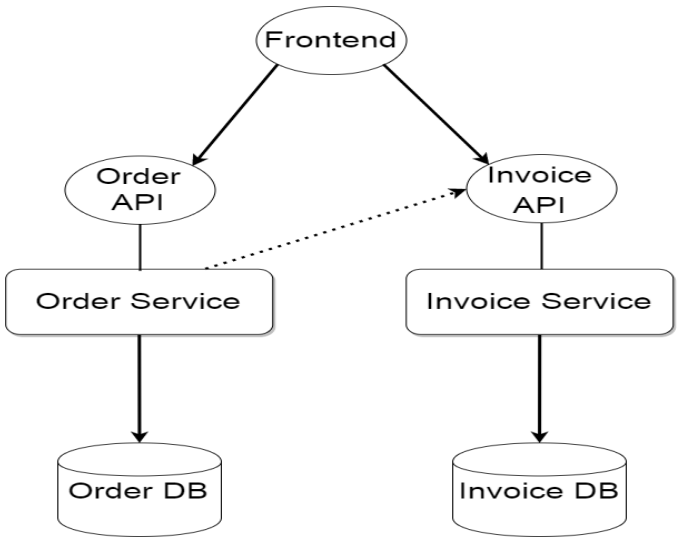

Figure 2: A small example of an application with microservice architecture.

Because of the small size and focus on only one business context, microservices makes it possible to achieve good modularity in the codebase. Modularity means designing the components of the application separately and independently [Bal00]. Modularity eases the development because it makes changes in one module independent from other modules [Bal00]. Modularity is easy to retain with microservices because there are clear boundaries between each of the services [Ric15c]. One microservice can only see the interface of other microservices, preventing calls to internal methods of other microservices. This means that accidental breach of the modularity is harder and keeping the clear modularity does not require discipline from the developers [LF14].

Clear modular boundaries with REST interfaces can introduce performance issues if the services are too fine-grained [Ric16]. If one business use case has to go through numerous service calls, the execution time of remote calls adds up. For example, if one call to a service takes up to 100ms, then calling 10 services takes one second. In this case the number of the inter-service calls has grown too big and it might make sense to reconsider the service granularity. One solution is to re-design and re-evaluate the service granularity by combining services together but still maintaining the principles of microservice design. Another solution is to use message queues and making some of the inter-service communication asynchronous if it is possible.

Microservices should always comply with the Single Responsibility Principle (SRP) [New15a]. SRP means "gathering the things that change for the same reason and separating those things that change for different reasons" [Mar09b]. SRP is one of the SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) principles which are the five basic principles of good object-oriented software design [Mar03]. With microservice architecture, it is easy to hold up to the SRP because the architecture style encourages the creation of small units. Modularity also helps with the SRP, because when the boundaries are clear, then the accidental breaking of SRP is unlikely.

When developing microservices, the goal is to have services that are loosely coupled and highly cohesive [New15a]. Loose coupling means that services should not know anything about the internals of other services [Pap03]. This is achieved with microservices as they have clear boundaries by nature and only communicate through interfaces that each microservice publishes. Cohesion can be described as the tightness of related features in different modules [Bri96]. When microservices are separated correctly they adhere to SRP and they have a single business context on which they operate. If these qualities are applied to microservices, it means that the microservices are also highly cohesive.

In order to achieve loose coupling with microservice architecture, it might make sense to duplicate some of the code [New15a]. Typically developers have been taught to follow the DRY (do not repeat yourself) principle [Hun00]. DRY states that the same code should not be repeated in the codebase but instead the code should be reused. This is good advice inside one microservices but when multiple microservices share same code problems can occur [New15a]. If one service requires a change to the shared code it means that all the services which use the same shared library have to be also updated and deployed. This means that the services are now tightly

coupled. The situation can be solved by rather duplicating the code. It gives the freedom for each of the services to be independent.

Loose coupling, high cohesion, SRP and modularity are all considered good qualities of software architecture. These qualities and patterns can be found from any well-designed application, but with microservice architecture, it is more likely that these patterns and qualities remain in the codebase during the evolution of software [New15a]. Typically when the size of the organization and the size of the codebase grows, it becomes harder to keep these qualities in the software [Ric15e]. Microservice architecture has two properties making it possible to keep these qualities in the codebase during the evolution of software: clear ownership of code [Nor03] and microservices have smaller codebases inside the services. Ownership means that one team owns the microservice. Small codebases inside the services make it easier to handle for developers.

Because of the loose coupling, high cohesion, SRP and modularity, development cycles can be fast [New15a]. Modifying functionality and adding new functionality inside a microservice can be relatively fast as the services itself are smaller and thus easier to understand. The complexity moves from inside the services to the surrounding communication [SMD16]. Because of the simplicity of the services, developers can develop new features faster and thus making the development cycle fast.

Microservices architecture requires CI and enables CD [BHJ16]. When there are hundreds of services, code check-ins to each service have to be automatically validated. CI pipelines ensure the validity of the new builds. Each service has to have its own pipeline in order to ensure fast feedback. CI pipelines should be pretty fast to run as each of the services is quite small and thus the amount of time it takes to run automatic tests should not be too high. Microservice architecture provides fast feedback for the developers in the form of CI pipelines. As there is only one team working on a specific service, it is clear whose problem it is when a build breaks.

A fast CI pipeline and small services enable CD [Nai16]. Because the changes are contained inside a service it should be clear what has changed and thus the amount of regression testing remains pretty small. If all the automatic tests pass and the test coverage is high, a deployment straight to production can be made with good certainty. Because microservices enable fast deployments and roll-backs in cases of faulty deployment, a problematic service can be rolled back in to a previous version. These qualities make CD possible.

CD enables fast release cycles where functionality that was just developed can be deployed to production during the same day. This gives organizations a chance to adapt faster to the needs of customers and also enables the possibility to experiment with new features [OAB12]. Using monitoring tools organizations can then figure out if the functionality should be expanded or if it is not even needed as customers are not using it.

Microservices together with CD enable blue/green and canary releases [New15a]. In blue/green deployments the new service is deployed on the side of the current version that is in production. Smoke tests that verify that the deployment was successful are ran against the new version and after they have passed, the production traffic is redirected to the new version of the microservice. The old version will still be on the side in case something goes wrong. Canary releasing means that a part of the traffic is redirected to the new service and the traffic and metrics are monitored. If something goes wrong, the traffic is redirected to the old service. In case that everything is normal the traffic is slowly moved to the new service and the old service will eventually be removed from production. Canary releasing makes experimenting with microservices even easier as the team can collect multiple metrics from the new version and thus decide if it satisfies the needs of customers better than the old service.

The complexity of microservices lies in the interconnections between different services [Fow15c]. Services themselves are small can be quite easy to understand but when business use cases require communication between different services, the complexity kicks in. For example, debugging calls that go through multiple services can be hard. This operational complexity is hard to handle and requires a lot of technical skills from the development teams. There are tools to help with the problems but still teams have to acquire new skills to handle these tools [Fow15c].

Microservices provide very fine-grained and flexible scaling [CJS16]. Different services might require different scaling. One service might require horizontal scaling while another one requires vertical scaling. Horizontal scaling means adding more instances that serve the microservice. Horizontal scaling can be automated by auto-scaling the components, which automatically adds more instances [TBB+15]. Vertical scaling means adding more capacity to the instance. Vertical scaling is not as elastic as horizontal scaling because adding or removing capacity to an existing instance requires downtime [RMMB15]. With microservice architecture, it is possible to scale every service separately depending on their needs whether it is horizontal

or vertical scaling that is needed.

One of the benefits of fine-grained scaling is spreading the risk by scaling[New15a]. The most critical services can be spread across multiple hosts, physical box or even data centers. With this kind of approach, the downtime of critical services can be minimized.

The microservice architecture enables polyglot implementations. Polyglot programming means that more than one programming language is used [WC10]. Teams developing microservices can make independent choices from other teams depending on their business and technical challenges. Some limitations should be in place, in order to limit the number of languages in the application. One possibility is to use a subset of polyglot approach, where only languages that execute on the same virtual machine are allowed [WC10]. Even if the number of programming languages is limited, the polyglot approach gives teams better tools to tackle the business and technical problems in their service [CJS16].

Microservice architecture gives the freedom to pick technologies based on the challenges instead of having to use the tools that were selected before. The polyglot approach also makes it easier to experiment with new technologies, because the business concerns are limited, the experiments have a small impact on the whole software [CJS16]. These experiments can then be shared with other teams if they are successful or discarded if they failed. The polyglot approach can also introduce performance improvements on services that require better performance [New15a]. A technology stack that improves performance can be selected for these services.

On top of the polyglot approach regarding programming languages, microservices also enable usage of multiple database systems [Ric15b]. The data that one service is handling can differ a lot from the data of another service. For example, one service might have data that fits perfectly to a relational database management system (RDBMS), while the other service has data that fits better to a NoSQL database such as Cassandra.

Microservices expose boundaries inside the application by having a lot of services with clear interfaces. This eases testing because there are more options on where to test and how to test [Cle14]. Small and modular services are easy to test because there are not a lot of dependencies that needs to be initialized. Also, cohesive services that implement bounded contexts means that the need for regression testing is reduced because the effects of changes are limited mostly inside the service.

Microservices are small by definition, even though there is no clear rule about how small they should be [LF14]. However, they should be small enough so that one team can have the responsibility of the whole microservice [New15a]. Team sizes can vary, but a maximum number of team members is considered to be around dozen people or as Amazon calls their service teams: "two-pizza teams", meaning that the size of the team should not be bigger than what two pizzas can feed [Mun15a]. Instead of trying to figure out whether the team is small enough, it can be easier to notice when the team size or the service is too big by closely observing the team communications and the problems that might arise [New15a]. When either the size of the team or size of the service is too big, then breaking up the service to two or more smaller services should be considered. When the service is broken into two services, it could also mean splitting up the team into smaller teams.

Companies such as Amazon[Mun15b], LinkedIn[Ihd15] and Netflix[Mau15a] have made the transformation to microservices. Their positive experiences and long-term usage with this architecture style have caught the interest of many other companies and developers interested in new architecture styles [Wol16]. These companies are very open about their development processes and, for example, Netflix has open-sourced a lot of their internal tools [Net15]. These open-sourced tools make it easier for companies who are taking microservice architecture in use to get started because they do not have to solve every problem by themselves. Instead, they can borrow best practices and use these open-sourced tools to solve the general technical challenges that adopting microservice architecture requires. This means that most of the development resources can be directed towards refactoring of their own existing codebase.

2.4 Main differences of monoliths and microservices

Monolithic and microservice architecture both have their pros and cons. Neither of them can be considered as a silver bullet for every organization [LF14]. Instead, the decision to go with a monolith or with microservices should be done case by case. Monolithic architecture has its own use cases while microservice architecture is the better choice for some applications. Monolithic architecture has been and is still the standard way to start application development. Even though microservices have gained popularity in the past few years the general consensus is still that it is better to start with monolithic architecture [Fow15b], however, there are also opposing opinions [Til15]. Organizations should evaluate their software architecture during

the evolution of software and pick the architecture that makes the development easier in the near future. It is possible to build monolith that is modular and has SRP in the modules [New15b]. With this kind of approach, the transformation to microservices can be easier later on down the road.

2.4.1 Development

It is easier to start developing applications with a monolithic architecture. However, as the code base grows in size, the problems of monolithic architecture increase [Ric15e]. Microservice architecture handles the increased size and complexity of codebase better [Fow15a]. Well defined and separated modules ensure that the complexity is better contained thus making the development faster.

Developing new features is typically considered easier the smaller the codebase is. Good modularity also helps developers to locate faster the correct place to apply the changes. Microservice architecture eases the development of new features when the complexity is high because each service has a small codebase and modularity is easier to retain when there are clear module boundaries [Fow15a].

In a monolithic application with a big codebase, it becomes slower to develop new features because the codebase is large and the modularity usually decreases [Ric15e]. Without clear modular boundaries it is very easy to accidentally tangle up different business contexts. Direct calls to inner functionality can be made unless strict restrictions are enforced. For example, in Java, it might be tempting to change the visibility of an inner method to public instead of going through the module interfaces. Monolith applications can also be modular, but it requires self-control and discipline to maintain the modularity in monolith applications.

Code duplication can be mostly eliminated from a monolithic application as the codebase is shared by following DRY principle [Hun00]. With microservices, it might make sense that the different services have duplicated code in order to achieve looser coupling [New15a].

Refactoring tasks in large monolithic codebases with tight coupling can be a daunting task [Fea04]. Changes can have an effect on multiple places and testing that nothing is broken can be a big task. Experimenting in a monolithic codebase is also hard, as refactoring and the introduction of new technologies has to be considered seriously. Microservices enable faster refactoring because the services are small and so even a complete rewrite of a service should not require a lot of work, as long as the interface

stays the same. This also means that experimenting with new technologies is a lot easier as changes are contained in small services.

Multiple different technologies can be used with microservices [New15a]. This polyglot approach gives better tools to solve different problems. Monolithic applications are stuck with the technologies that were selected before. For example, changing the programming language of a monolithic application's back end requires rewriting of the whole back end. These big rewrites are expensive. Rewriting a single microservice does not require a lot of work and the risks are much smaller than with monolith rewrites, as the problem space is limited within a service.

Using multiple different technologies requires new skills from the organization [BHJ16]. With monoliths, the number of different technologies is limited which makes it easier to develop expertise in these technologies. In order to support the polyglot microservices in production development teams have to have vast technological expertise instead of just focusing on a handful of technologies. If there are not enough people with appropriate skills either current developers have to be trained or new recruitments have to be made.

2.4.2 Database

Another problematic area regarding accidental tangling of business contexts with monoliths is the database layer. It is very easy to introduce accidental integration of different modules in the database layer. If the modules operate on the same schema, the easiest route for developers is just to change the data straight in the database layer instead of going through the interface of another module.

With microservice architecture developers can no longer access the database of another service directly. Because of the hard boundaries the accidental integration at the database level is not possible. Instead, the services have to go through the interfaces which other services provide.

However, transaction handling is a lot easier with monoliths because they use a single database. With microservices and multiple databases transactions provide complex challenges. Eventual consistency might have to be used instead of transactions. Other solutions are to use distributed transactions or compensating transactions which have their downsides.

2.4.3 Scaling

Microservices enable more fine-grained scaling compared to monoliths. Horizontal scaling by duplicating the application and data partition can be achieved with both approaches. On top of these, microservices also enable the scaling by functional decomposition. This means that with monolithic architecture the whole monolith has to be duplicated to new instances. With microservices, the most important services and the services which are under heavy load can be duplicated separately. With this kind of approach, the instances can be smaller as single services do not require as many resources as a whole monolith application does.

Figure 3 pictures the horizontal scaling of a simple monolithic application with four different business contexts. Every part of the application is scaled together. For example, now if the parts of the application which require more scaling are invoices and orders, then the whole monolith has to be deployed in order to get better scaling. Figure 4 pictures the horizontal scaling of microservices application with the same four different business contexts. With microservices, it is possible to have a different number of services deployed. In Figure 4 there are now four order services and four invoice services deployed which gives better scaling for these parts of the application.

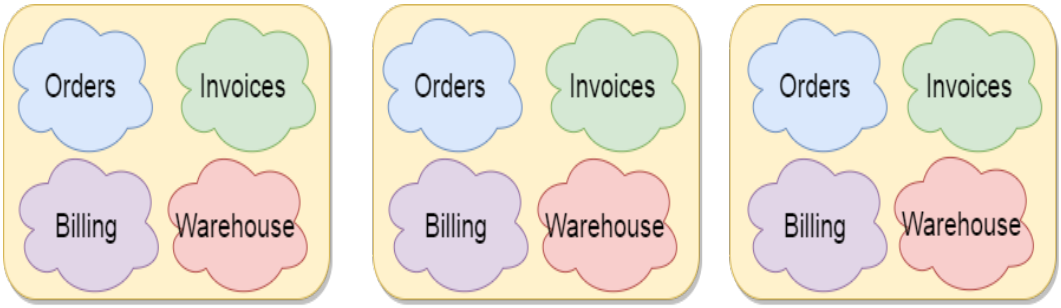

Figure 3: Example of horizontal scaling of a monolith.

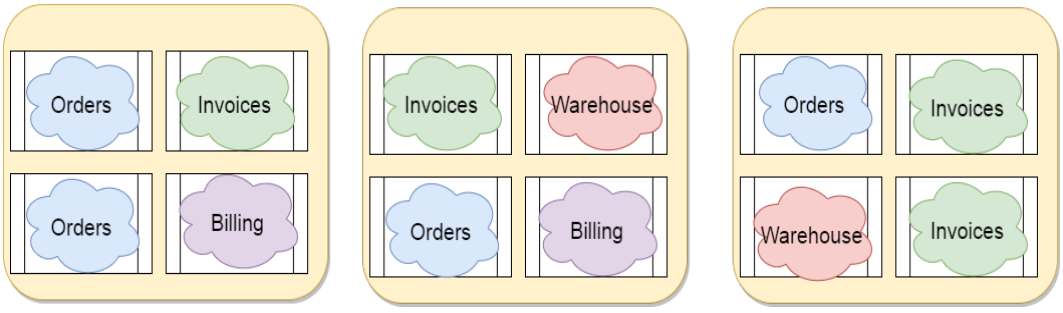

Figure 4: Example of horizontal scaling of microservices.

2.4.4 Production environment

Deploying monolithic application is simple [Ric15e]. There is only one artifact to be deployed. This simplifies the deployment process and the deployment is easy to automate. Microservices architecture provides multiple artifacts that need to be deployed. The number of services can be hundreds, which also means that the number of artifacts can be in the hundreds. With monolithic applications, it can be possible to handle some parts of the deployment manually even though it is not recommended. However, with microservices, it is essential that the whole process is automated. Microservices also aim to make frequent releasing easier. This can result in a situation where there are over thousand deployments in a day [Nov17]. That amount of deployments means that the whole process has to be automated. Rolling back faulty deployments has to be easy so that the user experience stays good.

Microservices require continuous and automated monitoring [TBB+15]. Service specific metrics have to be collected in order to automatically decide whether the deployment was successful and if the service is running properly. With a monolithic application, such fine-grained and extensive monitoring is not needed. Only one artifact is deployed and it is easy to monitor the state of that artifact. If one part of the monolith is not responding then it is very likely that the whole monolith is down. Monitoring of one application running in multiple instances is easier than monitoring multiple services running in multiple instances [New15a]. Typically with microservices also the number of instances is much higher than with monoliths.

Microservice architecture enables resilient applications by having multiple small services [TBB+15]. With a monolithic application the resiliency is harder to achieve as there is a single point of failure. When continuous monitoring is used together with microservices, the problematic services can be restarted automatically in seconds and end users might not even notice the problems that a service is having in the background. On top of the continuous monitoring circuit breakers such as Hystrix can be used to ensure better resilience [Net16]. If the monolith is experiencing problems and goes down then users cannot use the program at all before the monolith is back up.

2.4.5 Testing

Testing strategies and business use cases used in microservices and monoliths do not differ a lot. The best practices regarding testing strategies apply to both architectural approaches. Most of the focus on testing should be on fast running automatic tests such as unit tests [Cle14]. However, in order to get the most out of microservice architecture, the aim should be on continuously deploying new versions to production, which means that the changes have to be validated fast with automated tests. With a monolithic application, the release cycles are usually longer, which gives more time for long running tests and manual testing. With microservices, good automatic test coverage is a requirement unlike with a monolith application. Even though it is recommended to have good test coverage with a monolith it is possible to have smaller coverage and use more resources on manual testing. However, this leads to a slower development process.

Creating automated tests for microservices is easier because each service exposes new possibilities for tests [Ric16]. The services adhere to SRP and are small thus making it clear what is the purpose of a service. Inside a service, good coding standards should be followed by making the responsibilities clear for each method and keeping the size of methods small thus making the testing of them easy [Mar09c]. With a monolithic application, the risk is that over time methods and classes become too big and do multiple things. Big methods and classes with multiple responsibilities make testing harder [Mar09c].

Testing integration of multiple microservices locally at the same time has its own challenges. In an ideal situation most of the testing is done against one microservice but in some cases, the setting up of multiple microservices locally is required. A DevOps mindset and skills are required to painlessly achieve this. Tools such as Docker can help solve these challenges [JNS16]. With a monolithic application local deployment is easy and the support from IDEs is great [Ric15e]. Developers also have a lot of experience with running monoliths locally and using the tools that IDEs support.

2.4.6 Continuous integration and continuous deployment

Microservice architecture requires CI and CD to enable the frequent deployments of different services. Both microservice architecture and monolithic architecture can utilize CI. Using CD with microservices is a lot easier than with a monolith. Even

though CD is possible with monolith it requires a lot of work and CD is seldom used. With the CI and CD pipelines microservice architecture enables faster release cycles than monolithic architecture. This means that the organizations utilizing microservices together with CD can react faster to the needs of customers than organizations with monolithic application that does not utilize CD.

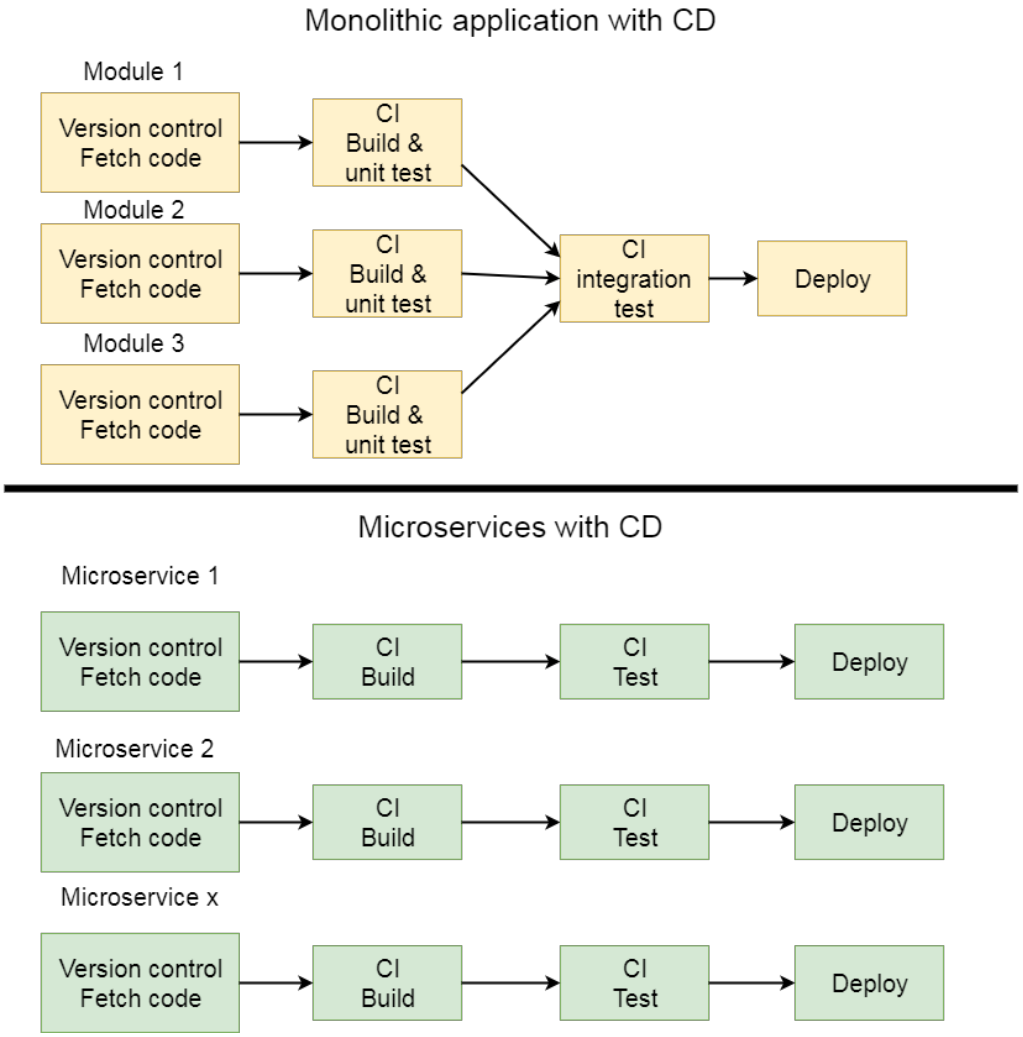

Figure 5: CD pipeline comparison between monolithic application and microservice application [BHJ16].

Figure 5 illustrates the main differences when using CD with a monolithic application compared to using CD with microservices. As we can see the CI test phase can become a bottleneck when using monolithic architecture and CI. Also the deployments are much bigger with monolithic architecture as the whole application is always deployed compared to microservices pipelines which deploy only a single service.

Category	Monolith	Microservices
Time to market	Fast in the beginning, slower later as the code base grows.	Slower in the beginning, because of the technical challenges that microservices have. Faster later
Refactoring	Hard to do, as changes can affect multiple places.	Easier and safe because changes are contained inside the microservice.
Deployment	The whole monolith has to be always deployed.	Can be deployed in small parts, only one service at a time.
Coding language	Hard to change. As code base is large. Requires big rewriting.	Language and tools can be selected per service. Services are small so changing is easy.
Scaling	Scaling means deploying the whole monolith.	Scaling can be done per service.
DevOps skills	Does not require much as the number of technologies is limited.	Multiple different technologies a lot of DevOps skills required.
Understandability	Hard to understand as complexity is high. A lot of moving parts.	Easy to understand as the code base is strictly modular and services use SRP.
Transactions	Easy to use ACID transactions supplied by the RDMBS.	Hard to implement. Eventual consistency has to be agreed on some cases
CI, CD	CI is possible and should be used. CD is hard to achieve	Using CI is required and CD should be used. CD enables faster release cycles.

Table 1: Comparing monolith and microservices

2.4.7 Summary

Table 1 on page 21 contains a conclusion of the comparison between these two architecture styles. From the table, we can conclude that both of these approaches have their pros and cons. So deciding which one to use depends a lot on the challenges that the organization is facing.

As a conclusion, the monolith is faster and easier way to start developing new applications. When the complexity of the system raises, the codebase grows bigger and the size of the organization grows, this is when microservices become more and more attractive. A general observation could be made that microservices are not a good choice for organizations which are small. Also, organizations that do not yet have a good understanding of the business context in which they operate should not start with microservices because it is expensive to get service boundaries wrong [New15a].

It seems that it is popular to start with the monolith and then make the transformation to microservices when the complexity and size factors kick in. Companies such as LinkedIn[Ihd15], Netflix[Mau15a] and Amazon[Mun15b] have followed this route. After these companies had gained enough traction and their businesses had grown to a size where the monolith architecture did not provide enough scaling and separation, they made the transformation from monolith to microservices.

3 Challenges when transforming from monolith to microservices

Microservice architecture provides multiple different challenges. These challenges can be separated to architectural and organizational challenges. In this work most of the focus is on the architectural challenges. One of the biggest challenges that the microservice architecture presents are database related challenges.

3.1 Database challenges

The first challenge with databases is to select the correct database pattern. There are two prominent database patterns, database per service and shared database. When each service uses its own database the pattern is called database per service [Ric15b]. This pattern is preferred for microservices because then the database is

not a point of integration for the services [New15a]. Instead, services have to use the APIs published by other services to access the data that they need from other services. With the database per service pattern, microservices stay loosely coupled. This pattern however brings challenges with the transaction handling.

The shared database pattern has multiple downsides [New15a]. When the database is an integration point it can become one huge, shared API for the whole application. Changes to the schema require a lot of regression testing and making sure that nothing is broken. Shared database means that the database engine that is used has to be same for every part of the application. This limits the polyglot approach as all of the data lives now in the same database even though some parts of the application would benefit from a different kind of database.

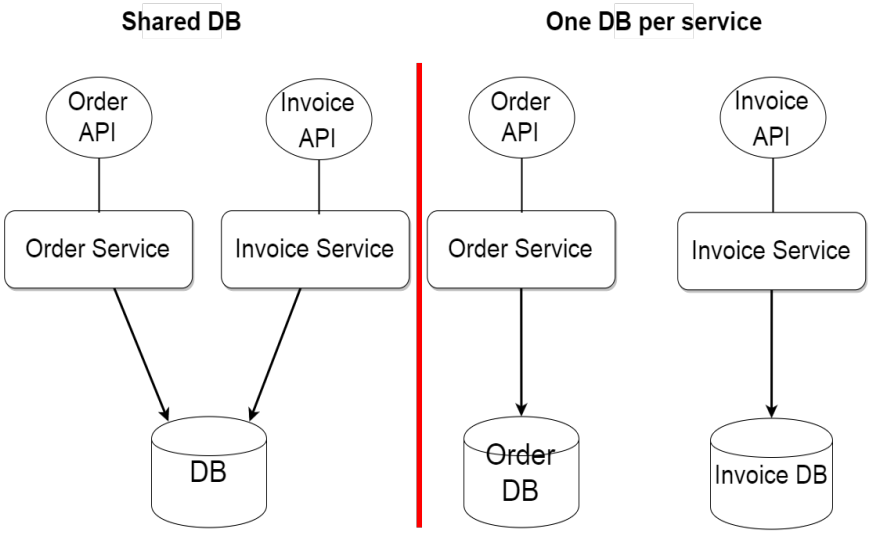

Figure 6: Shared database and database per service patterns [Ric15b]

In Figure 6 we can see two different ways to handle the data with microservices. The example on the left side uses the shared database pattern [Ric15d]. On the right side, there is an example of database per service, which is a much better pattern as it keeps the data separated and gives loose coupling between the services [New15a].

With Database per service pattern, services cannot access the same database anymore. It means that the foreign-key constraints and database accessors accessing data that belongs to another bounded context have to be split up. Instead of accessing the data straight from the database calls to separate services should be made. For example, if an invoice has a link to an order in the database and the name of the order is then queried from order table this query has to be replaced with a call to the order service to get the name. In Figure 7 on page 24 we can see an illustration

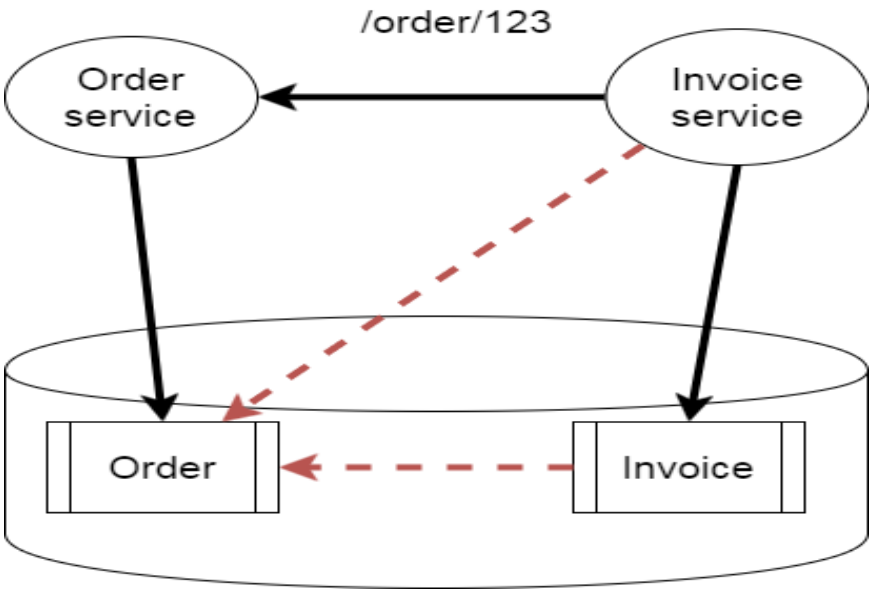

Figure 7: Example of refactoring the database layer to use database per service pattern.

of this example. The straight queries and foreign key constraints to the order table have been cut and the invoice service now goes through the order service API to fetch the information it needs about the order.

On top of the splitting of foreign key constraints and straight queries transactions pose a big challenge with microservice architecture. When there are multiple databases using transactions is not possible anymore. Transactions ensure, for example, that either all of the data is inserted to the database or none of the data is inserted. Transactions are a very useful tool for developers. Microservices utilizing multiple databases take this tool away from developers. Developers have to replace the transactions with either eventual consistency, compensating transactions or distributed transactions [New15a]. Each of them have their own downsides and use cases.

Eventual consistency means that the failed operation will be tried again later. A queue can be used to store the failed attempt and at a later time, the operation can be retried. For example, if an insert to the order table is successful and the business use case requires also an invoice to be created from the order. Then, if the creation of the invoice fails, we can queue the needed information about the order and create the invoice at a later time when the service is operational. This means that the system might be in an inconsistent state for a while but eventually it will

get back to a consistent state. The eventual consistency approach is useful with business operations that are long-lived [New15a].

Another option is to use compensating transactions. Compensating transactions means that if one of the later operations fails, all of the operations will be rolled back. For example, if the insert to order table was successful but the insert to invoice table failed, then the insert to order table will be rolled back. A delete statement will be issued for the inserted data in the database and the user will be informed that the operation failed. This approach has also its downsides, it might be hard to decide where the logic on doing the compensating statements lies and what happens if the compensating transaction fails. With compensating transactions the handling of transactions can become hard to understand and implement [New15a].

An alternative to compensating transactions is to use distributed transactions [New15a]. Distributed transactions use a transaction manager to handle the transactions. Distributed transactions try to handle transactions just like in a monolithic application but with multiple databases and over the network. Typically they use two-phase commit in order to ensure transactional safety [New15a]. This approach has its downsides from the needed locking which can make the scaling of systems harder. Scaling is one of the main pros of microservice approach so using distributed transactions can diminish the returns of the microservice approach.

All these different ways to handle transactions have their downsides and deciding which one to use should be done case by case. Using eventual consistency when it is possible is advised [New15a]. Eventual consistency makes things simpler and does not limit the scaling possibilities. If there is a situation where eventual consistency is not applicable, then it should be seriously considered if it even makes sense to split these two things up to their separate microservices.

Regardless of the selected way to replace transactions microservices have to be fault-tolerant by nature [MW16]. Especially, when the number of services grows, so does the possibility that one service might not respond. Scaling microservices to different hosts is not enough. The service can be under excessive load or completely down. For these cases, the circuit breaker pattern is needed. Circuit breaker handles failures fast and can provide a fall-back which returns default data instead of waiting for the response from a dependency [MW16]. With these methods, microservices can enable resilient application that recovers timely from failures and end users might not even notice the failures.

3.2 Microservice premium

Microservice architecture introduces technical challenges that are not present in monolithic architecture [New15a]. This is the so called microservice premium [Fow15a]. It means that when using microservices there is additional complexity introduced to the system. The additional complexity comes from various sources such as rapid provisioning, needs for monitoring and rapid application deployment [Fow14b]. On top of these microservices introduce significant operational overhead, distribution and asynchronization which are complex problems to solve [Woo14].

So, if the organization decides to start a new project with microservice architecture, it needs to spend time solving these challenges before it can start the development of features. This means that the initial development of the application is slower with microservices than with monolith approach. As the complexity of the application grows microservice approach begins to catch up with the monolithic approach in terms of productivity and eventually the productivity becomes better with microservices [Fow15a].

3.3 Communicational challenges

Communication between different services is an important part of microservice architecture [New15a]. There are multiple different approaches on how to handle the communication between different services. One of the most popular ones is that every service publishes a REST API over HTTP. Companies such as LinkedIn use this approach [Ihd15]. The APIs must be as backward compatible as possible to ensure that the clients depending on the APIs do not require continuous changes. REST APIs are a good choice because they are familiar to developers, they are language independent and enable backward compatibility.

However handling all of the communication over REST APIs is not a good solution [Ric16]. For asynchronous communication using messaging instead of REST APIs provides better performance and reliability [Ric16]. Message brokers like RabbitMQ can provide message queues [Piv17a]. One service will publish events to this queue and another service can subscribe to these events and handle them when it can. Messages are persisted in this queue which increases reliability as the service who publishes events can forget about them as soon as they are published. Message queues can add complexity to the development but once they are up and running, they ensure the loose-coupling and event-driven architecture [New15a].

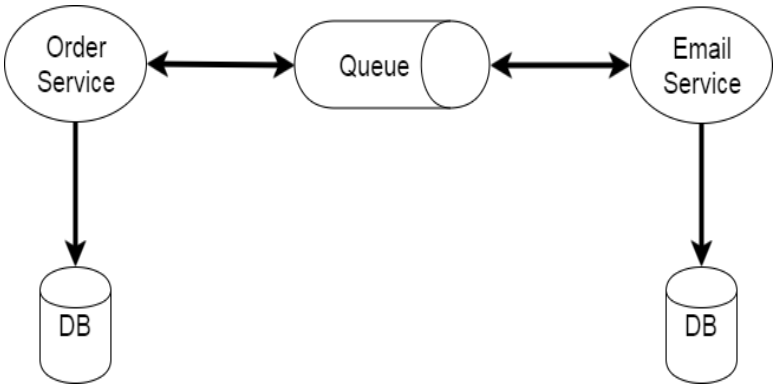

Figure 8: Message queue where events can be published and services can subscribe to events

Figure 8 has an example about the message queue solution. Order service can publish events about the orders it receives to the queue. Then email service can subscribe to these order events and after a new order is received, the email service can then send an email about the order to the customer. With the message queue, these two services are separated from each other and their communication can be asynchronous. The order service does not have to wait for the email service to send the email and return a response to the order service that the sending of the email was succesful. Instead the order service can just publish the event and continue. The email service will eventually handle the new event from the queue and the user will receive an email about the order.

New boundaries and communication between various services can hinder the performance of the application [New15a]. With the microservice approach, the number of network calls increases. Because of the separation of the data, the number of database queries can increase.

3.4 Testing challenges

Testing microservices also presents its own challenges. Even if testing is easier inside one microservice, the testing of the whole system can be a complex problem. Microservices introduce more moving parts, which means that ensuring that the tests are comprehensive enough is not easy [JNS16]. Testing a distributed system with multiple independent components is hard. On top of the normal unit tests, integration tests and end-to-end tests that can also be found from a monolithic application, microservices also require tests for the components and the contracts

that the services provide.

This means that validating the behavior of microservices requires multiple different testing strategies [Cle14]. These testing strategies are unit testing, integration testing, component testing, contract testing and end-to-end testing. Like with any software architecture most of the focus should be in the testing strategies which are fast to run and easy to maintain. Microservices introduce multiple boundaries between different services, which means that the interaction in these boundaries has to be well tested in order to ensure that the whole system works correctly. This means that extra attention has to be paid for contract testing in order to validate the communication between different services.

Microservices also require performance tests in order to make sure that the new boundaries and communication between different services does not slow down the application too much. Performance testing should match the traffic of the application in production and possibly even more to see that if the traffic increases, no problems occur. There should be performance tests of different granularity. Some performance tests which go through the whole end-to-end service calls and some performance tests for single services.

3.5 Observability challenges

Microservices require continuous observing [JNS16]. With multiple different services running, problems are more likely to occur and the service causing the problems has to be located fast. Because of this there is a need to visualize the health status of every service in the system [JNS16]. To solve this challenge aggregated logging and continuous monitoring of every service is needed.

When there are multiple services and servers running in production, it is not feasible to manually monitor them and browse the logs from each server [New15a]. Microservices require an aggregated logging system, which makes viewing logs possible from every server and service in one centralized place. There are tools such as Logstash[Ela17b] and Kibana[Ela17a] that handle the aggregation of logs.

Running numerous distributed services in production requires continuous monitoring [TBB+15]. Metrics from every service has to be collected. These metrics have to be monitored in order to respond fast to possible error situations. Metrics such as CPU load, response time, the number of errors and service usage can be collected [New15a]. These metrics can then be used to set up alerts when there is something

extraordinary.

Multiple distributed services poses also the challenge of service discovery on top of the observability [SMD16]. When the number of services grows, it becomes harder to know what service is running at a specific environment. Tools such as Docker[Doc17] can simplify the problem but it is still a complex challenge that requires resources and tooling to solve the problem. Service discovery tools should provide the services functionality to register themselves and a way to find services once they are registered [New15a]. There are multiple tools to handle this challenging problem. Tools such as Eureka[Net17], Zookeeper[The17] and Consul[Has17] help developers solve the problems of service discovery.

3.6 Organizational challenges

On top of the technical challenges microservice architecture requires a different kind of organization structure compared to monolithic architecture [New15a]. Organization structure needs to adapt together with the architectural changes. The structure of the organization has to be similar with the structure of the application [New14]. This means applying Conway's law to the organization. Conway's law was presented in Melvin Conway's paper *How Do Committees Invent* and states the following: "Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure." [Con68]. This means that organizations have to order their structure towards independent teams that can develop, test, deploy and take care of their service in production.

Teams are responsible for every part of a microservice from development to running it in production [Mau15b]. Teams have a lot of responsibility but they have also a lot of freedom. This kind of approach gives the power to the teams. They can decide what tools to use during development, which programming language suits best to their area of concern, what kind of database is needed and so on. Teams also have full control of their microservices codebase. Ownership of codebase and the whole microservice improves the quality of the application [Nor03].

Even though teams have a lot of freedom, it makes sense to set some boundaries for the teams. If there are no limits to the freedom, the application might end up with too many different technologies which can lead to situation where there is not enough expertise for each of the technologies [Fow15c]. This limitation can

be supported by providing the common tools such as monitoring only for a set of technologies. Organizations can also give some rough guidelines to the teams such as that all the services have to run on Linux platform. However, the restrictions should not go too far. Experimenting should not be restricted and trying out new technologies should still be encouraged [Fow15c].

The sheer number and complexity of these challenges makes it clear that the transformation from monolith to microservices requires a lot of work and microservices cannot be considered as a free lunch.

4 Techniques for transforming monolithic to microservices architecture

Transforming an existing application from monolithic to microservices architecture is not a small task. There are various enablers before the transformation from a monolithic application to microservices is possible.

After the refactoring enablers are fulfilled, the decomposition process towards microservices architecture can begin. If the codebase of the application is big, the transformation process should be carried in small steps in order to get the microservice architecture right without too big risk. The first step should be to pick the right parts of the codebase to be decomposed. There are multiple criteria on what functionality should be refactored next. The criteria are such as coming functionality changes, new functionality to be added, the least tangled parts of the application and the need for new technologies or programming languages in the application. For example, there might be parts of the monolith that would benefit from using a different technology or programming language and thus it might make sense to decompose these to a microservice to enable the polyglot approach.

After the functionality that will be decomposed is selected, the separation process can begin. The functionality and data should be separated from the monolith and communication between the monolith and microservice should be handled with an anti-corruption layer.

4.1 Enablers

Before the transforming can begin various enablers for the transformation process have to be fulfilled. These decomposition enablers are such as good test coverage, buy-in from the whole organization and modular monolith. Implementing these enablers can take time depending on what is the situation of the codebase regarding testing and its modularity. Also the organization's technical skills have a factor on how long it takes before the transition process can begin. One factor is also, whether the developers are familiar with microservices and the technological challenges that come with them.

4.1.1 Testing

Before the refactoring towards microservices can begin, the monolithic codebase has to have good test coverage in order to safely carry out the refactoring. If the test coverage is lacking it is very likely that new bugs are introduced during the refactoring process. If most of the testing is done manually, it makes sense to first get the automatic test coverage up for the part that is going to be transformed to microservices. Microservices benefit a lot from the automatic test coverage because with good test coverage, they can be released frequently. Various testing strategies such as unit testing, integration testing and end-to-end testing should be used in the monolith. These tests give the certainty that the functionality behaves the same way when the refactoring to a microservice has been done.

Figure 9 on page 32 shows the testing pyramid, which illustrates how the amount of testing should be divided. Most of the tests should be unit tests which are fast to run and give quick feedback to the developer. The scope of the unit tests is small by definition. Integration tests give better confidence than unit tests, but they are slower to run. End-to-end tests which are for example UI tests provide the best confidence, but they are very slow to execute and likely to break. If the monolith has good test coverage and the number of different tests is distributed the same way as in the pyramid, it gives better certainty to the developers that their refactoring was successful.

If the service that is decomposed uses the same programming language as the monolith, then unit tests that were written against the monolith should be easy to refactor together with the separation of the microservice. These tests still ensure that the smaller units work as they should even after the decomposition of the microservice

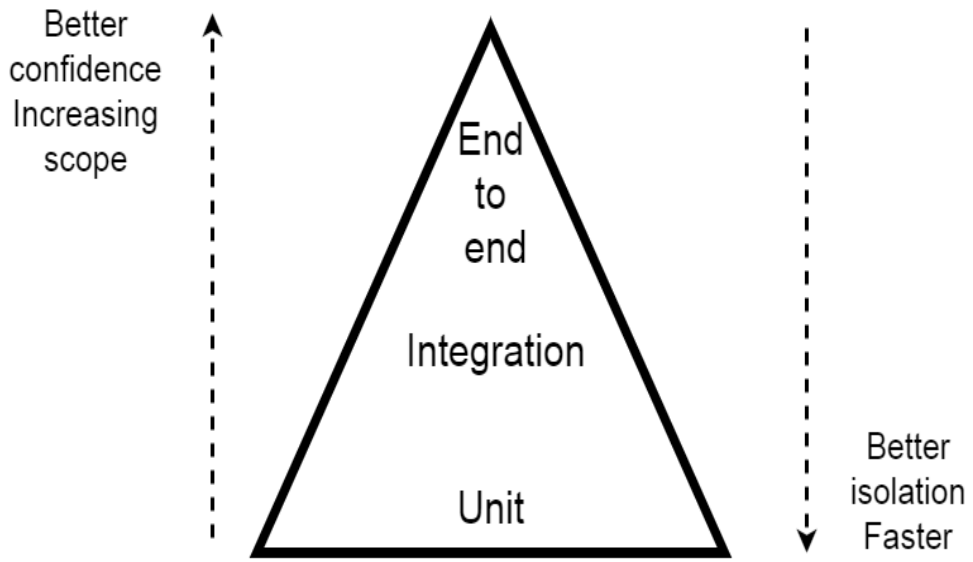

Figure 9: Testing pyramid [Coh10].

[Cle14]. They are useful also when the development of the new functionality continues inside the microservice. Even if the new microservice uses different programming language than the monolith, unit tests that were developed for the monolith can be used as a starting point when starting to write unit tests for the microservice.

Integration tests that are testing an interface inside the monolith can be used as integration test for the new microservice or used as a starting to point to write contract tests for the new microservice [Cle14]. These integration tests require some modifications as the interface can now be a REST interface but the business rules that were tested in the integration tests should still remain the same. From the business point of view, it is likely that the microservice will have a similar interface as the interface that was under the integration tests before in the monolithic codebase. Because of that, these tests will ensure that the microservice works as intended even after the separation and the functionality has not changed during the refactoring. For example, if the monolith used to have an interface for creating invoices, then, when the decomposing is done, the microservice will also have the same business rules and requirements for creating the invoice.

End-to-end tests cover a lot of the codebase and give a good certainty that a given functionality is working [Cle14]. This is especially useful when the functionality under testing is being decomposed to microservices. The team that is doing the decomposition can rely on the end-to-end tests. However, when the functionality under testing utilizes microservices, it means that also all of the microservices has to

be running. This adds complexity especially if the monolith also has to run together with the microservices.

4.1.2 Organization

Decomposing a monolith to microservices is a big architectural change which requires buy-in from the whole organization for a long time. If the monolith has a big codebase then the refactoring phase can take years [Sti15]. During this time two different codebases have to be supported, and at the same time teams have to learn a lot of new skills to be productive with the microservice architecture.

In the beginning, microservice architecture requires a separate team to handle the technical challenges that the new architecture provides. This platform team supports the teams building microservices by providing infrastructure to build the microservices [Mau15a]. It solves problems such as building the message queue architecture and developing templates for monitoring, logging and so on. These functionalities are not service specific, so it does not make sense for every team to build their own version of these. Even after the migration to microservices has been completed, this team should still exist in order to support the cross-cutting needs of the teams building microservices.

Initially, the development of new features might be slower as teams have to get used to the new architecture and the development process of microservices. Especially, if the functionality that has to be changed is still inside the monolith, a refactoring process has to be made to microservices before the changes to the functionality should be applied. This delays the release of the functionality.

Besides the technical challenges that the organization faces at the beginning of the refactoring, there are organizational challenges also. These organizational challenges vary from the structure of the organization to the skills of the organization.

Microservice architecture requires new skills from the teams [BHJ16]. Training and new recruitments might be needed in order to support the new architecture. With a monolith, the number of technologies was limited but this limitation goes away with polyglot microservices. Expertise on these new technologies has to be studied and distributed across teams which use these technologies.

The structure of the organization has to align with the structure of the application architecture [New14]. This means that if the teams were previously constructed based on the roles of people, with microservices teams have to consists of people

with different skill sets. Teams have to be formed in a way where they can develop, test, deploy and operate a service in production. Teams have to have cross-cutting skills and autonomy to make the microservice architecture work.

4.2 Techniques

When the organization has decided that it wants to transform its existing architecture from monolithic to microservices, the first idea might be that it is best to transform an existing functionality to microservices. However, it is better to start with a new functionality and build that as microservice [Cal14]. Building new services from scratch is a lot easier than refactoring existing functionality to microservices [Sti15]. All new functionality will be built as microservices and nothing new is added to the monolith. Slowly the number of microservices will grow. This kind of incremental approach is the best way to handle big architectural refactoring [New15a]. Organization and teams building these new microservices will gain knowledge about the microservice architecture and the challenges it presents. The core functionality of the application will still be served from the monolith which means that the starting pains with microservices are not noticed in the most important features of the application. When there is enough knowledge about how to build the microservices, then the organization is in a better place to start the refactoring of existing functionality to microservices.

The first problem is where to start with the decomposition. There are multiple factors which help to decide which parts of the monolith should be refactored to microservices first [Sti15].

The first job is to find the seams of different bounded contexts in the application [New15a]. Every business domain has multiple bounded contexts. These bounded contexts contain models which some of them are internal models which should not be shared with other bounded contexts. The bounded context contains also models which are shared through the interfaces. These different bounded contexts can be broken apart from the monolith to smaller pieces. These smaller pieces make up for good candidates to refactor to microservices. There should be an understanding about the different bounded contexts already in the organization as the existing monolith has been built and thus the problem space is familiar.

One example of bounded context from Procountor is Business Partner. A company can have three different partner types: customer, supplier and person. These

business partners can then be linked to invoices or other different things. Business partners contain information about the partner and different information is stored based on the type of the partner. Business partner makes for a good example about bounded context because even though they can be linked to invoices, they still contain their own business logic and data.

When the bounded contexts of the application have been identified, the next step is to decide which one should be refactored to microservices first. One way to find good candidates for the next iteration of transformation to microservices is to look at what pieces of functionality are going to be changed most in the near future [Sti15]. Microservices enable the fast rate of change so if the existing functionality is first refactored to microservices then the following development of these functionalities is going to be easier and faster. These new microservices can be deployed separately which speeds the feedback and helps to experiment with the new features.

On top of the fast experimenting, also new functionalities that would require a new technology to be introduced to the codebase are good candidates to be split up from the monolith. After the functionality is decomposed to a separate microservice, then these new technologies could be experimented with. Experimenting is possible as microservices enable polyglot implementations.

The team structure of the organization can also be a deciding factor on which functionalities should be separated [New15a]. If the teams are located in different geographical areas and thus their communication is slow and fine-grained communication is hard. Then it makes sense that the codebase that the teams work on is separated. This gives the ownership of the separated services to these geographically separated teams and lowers the amount of communication that they need to have between each other. The ownership and lowered amount of communication ease the development as teams can now make independent choices regarding their microservice. Even though this kind of separation is possible with monolithic architecture, microservice architecture reduces the amount of communication required even more.

Functionalities that are least tangled with the rest of the codebase are easier to refactor to separate microservices [New15a]. If there are already parts of the application which are loosely coupled and well separated from the rest of the monolith, it makes sense to refactor these to microservices. The cost and risks of refactoring are small. This separation ensures that the loose coupling and strong cohesion remains in the functionality.

After the candidate functionality to be transformed to microservice has been found

and all the decomposition enablers have been fulfilled, the functionality can be refactored from the monolith to microservice. There are various tasks that have to be done in order to make the refactoring.

The first step is to modularize the code inside the monolith. Qualities such as separation of concerns and cohesion should be introduced to the codebase in order to make the decomposition possible. The goal is to introduce these two qualities to the codebase of the functionality while still having the functionality inside the monolith. After this step, there should be a clear interface which could later represent the interface of the microservice.

Separation of concerns means that the basic functionality should be separated from the code that handles special issues such as synchronization [HL95]. Refactoring such as extract class can be used to achieve separation of concerns [FB99]. Extract class refactoring should be used when there is a class that has too many responsibilities [FB99]. This class should be split up into multiple classes. If the codebase already has a well defined layered architecture with, for example, Model-View-Controller architecture, the separation of concerns might already be at a good level and this step is already done.

The cohesion of the functionality can be achieved by various encapsulation related refactorings [FB99]. Hide method refactoring can be used to clarify the interface that the functionality should support. A method that is not used by anything else than the code that is related to this functionality should be declared either private or package level visibility [FB99].

After the functionality is separated from the rest of codebase and the functionality is highly cohesive with a clear interface the next focus should be on separating the concerns on database level. Typically database has a lot of tangled dependencies [New15a]. After the code has been separated to clear packages with bounded contexts, the same should be done for database accessors. With a monolithic codebase it is very likely that there are foreign-key constraints and database accessors access directly data of another context. In these cases, the database has become an integration point. This is something that should be avoided with microservices so refactoring of the data access layer and database structure is needed.

Data migration is always an expensive process. When the organization is starting the process of decomposition, it is very likely that the services change a lot and service boundaries can change. This can result in multiple data migrations. For example, when a service is too coarse-grained it has to be split up into two different

services. This means that the data has to be split up again. Another case is that the service is too fine-grained which requires the data to be joined together to one database [Ric16]. In both of these cases, two data migrations are required, one for the original split up from the monolith and another because the service granularity is not correct.

Expensive data migrations can be avoided by separating the functionality first and let the data still reside in the same database [Ric16]. When there is more knowledge about the services and the wanted granularity, then data can be separated. This approach gives more freedom to make changes to service interfaces and the extra data migration can be avoided.

A monolithic application typically utilizes ACID transactions in order to ensure that either all of the operations succeed or none of them happen. After the data has been migrated, transactional safety is lost [New15a]. This is because of the multiple databases that the separate microservices have. Thus the developer cannot anymore use transactions easily. For example, if an order is created and then an invoice has to be created based on that order, before with a single database it was possible to just use a transaction and ensure that both of these operations succeed or neither of them will happen. When decomposing the invoice or the order microservice from the monolith, the developer has to take this old transaction into account. Developers have three options to choose from: using eventual consistency, compensating transaction or distributed transaction. All of these have their downsides and it has to be considered on a case by case which suits best. The transaction changes have to be done in the monolith also if, for example, the order logic is now in a microservice and the invoice logic is in the monolith. Handling transactions in microservice architecture is hard and it is critical to get it correctly done.

When the functionality and the data are separated from the monolith it is still likely that the microservice has to communicate with the monolith. This when then anti-corruption layer should be built between the monolith and the microservice.

Building new functionality as microservices and refactoring existing functionalities to microservices requires that the microservices communicate with the existing monolith [Sti15]. The monolith might have data structures and coupling of data that is not wanted to creep into the microservices. In order to stop this from happening an anti-corruption layer can be built between the existing monolith and microservices [Eva04].

The anti-corruption layer is an isolating interface which communicates with the

monolith and offers functionality for the microservices to interact with the monolith. This interface sends information about the events happening in the monolith to microservices which need to act on these events. The preferred way to build this layer is to have a facade in the monolithic application that does not change the existing functionality of the monolith and at the same time offers decoupled communication between the monolith and microservices. The facade provides a simple interface to the monolithic application [Eva04]. It hides the complexity of the monolith and thus makes it easier for the microservices to communicate with the monolith.

Figure 10 illustrates an example of the inner design of an anti-corruption layer. The layer protects the microservices from the complicated interfaces and the data structures that reside in the monolith. The data structures might not be optimal for the microservice so they cannot be used as such. Instead, the anti-corruption layer provides data in the format that is required by the microservices. Services are clean interfaces which the microservices can use to send or receive data. The adapter allows microservices to use a different protocol such as REST to query the anti-corruption layer which can then handle the transformation of the communication protocol to the one required by the monolith [Eva04]. The conversion of the data received or sent to monolith is done by the translators. Converting the data to the needed format might be a complex task which is why the logic is separated from the adapters [Eva04].

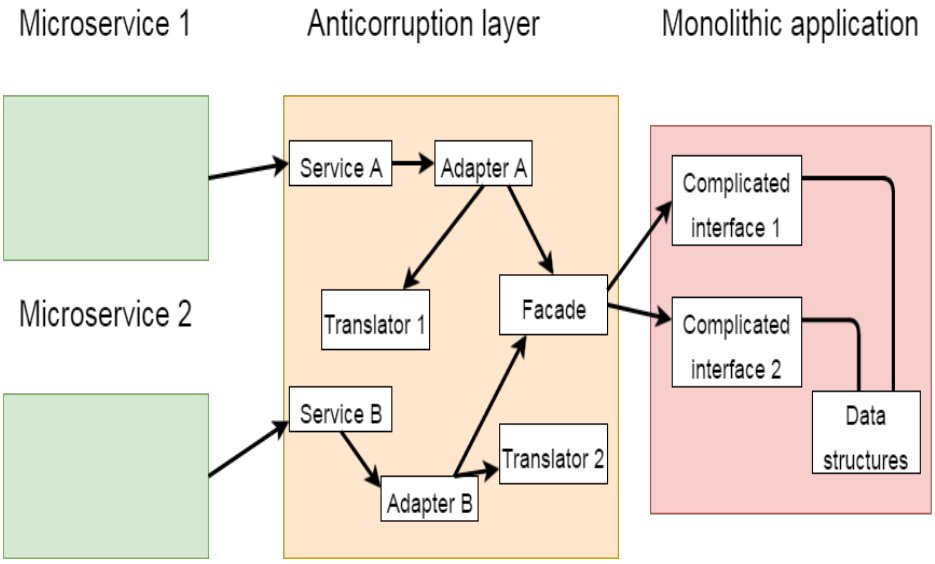

Figure 10: Illustration of the inner functionalities of anti-corruption layer [Eva04].

In Figure 11 we can see an example of anti-corruption layer between the monolith and microservices. The anti-corruption layer enables the communication between microservices and monolith. For example, if the functionality about creating orders is still in the monolith and the functionality for creating invoices from the orders is already refactored to microservices. Then the invoice microservice needs information about the orders. The anti-corruption layer will relay the information about created orders to the invoice microservice. The flow of the information can go also another way around as we can see from the Figure 11. Microservices might have events that require actions from the side of the monolith and this communication is done through the anti-corruption layer.

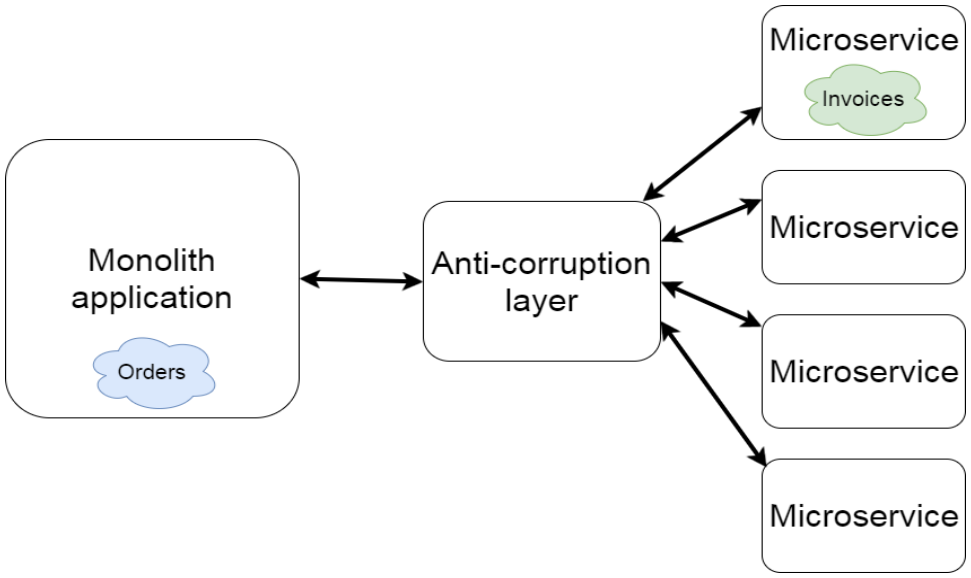

Figure 11: Example of anti-corruption layer between monolith and microservices. With the orders and invoices example illustrated.

After the anti-corruption layer is also in place the microservice can now function separately. The service granularity should be correct and all the good parts about microservices are now enabled for the service. This same process should then be repeated for other bounded contexts and new functionality. The services to be decomposed next can be picked up on the criteria that were explained before. Slowly most of the codebase should be in the microservices instead of the monolith.

Slowly transforming the architecture towards microservices by strangling the monolith gives the organization space to make mistakes and learn more about microservices [Sti15]. Strangler pattern is a way to slowly make the transformation and letting the monolith codebase to reside with the microservices for years if needed

before the full transition is made [Fow04]. The risks are much lower with this approach instead of going with a big refactoring at one time. Microservices which are separated and released can take advantage of the shorter release cycles when the monolith only gets critical bug fixes [Fow04].

The transformation process can be considered done when either the whole codebase has been transitioned to microservices or the monolith only has functionality that does not change very frequently [Sti15]. In some cases, it does not make sense to transform everything to microservices if the functionality that is left in the monolith does not benefit from the decomposition to microservices in any way. It might be enough that the anti-corruption layer is between the remains of the monolith and the microservices. There might be, for example, some functionality that has not been changed in years and transforming it to microservices might just introduce new bugs.

5 Transforming an existing application towards microservices

Microservice architecture provides multiple benefits compared to monolithic architecture in large applications. However, the transformation provides also multiple challenges. In order to get a better picture about these challenges and how hard they are, a case study was made. The case study examines the possible transition of Procountor[Pro17] core software from monolithic architecture towards microservices. The challenges that this transition provides might be easier to solve than the challenges that the monolithic architecture currently poses.

The case study was done by transforming one existing bounded context to microservices and evaluating the time spent doing this and the rewards that the transformation provides for the future development of this context. Also, a plan how to transform the more tangled bounded contexts was made. This was done by inspecting one of the most tangled parts of the application and figuring out the needed steps to transform it to microservice. A very rough time estimate about the future transformations and the required resources were made based on the transformation of the one service and on the plan to transform tangled parts to microservices.

Because of the large size of the Procountors codebase the resulting time estimate of this case study is only a rough estimate. Doing a full transformation requires changes

in the whole development organization and thus the transformation contains also other challenges than just the technical challenges that are focused on this case study. These challenges such as the structure of the organization and the required training together with the change of mindset towards DevOps based development has to weighed also in when making the decision whether to transform towards microservices.

5.1 Case introduction

This case study is done on the Procountor core product which is a web application for accounting and financial management. The system has been under development for over 15 years. The Front end has been transformed from a Java applet to a Vaadin [Vaa17] front end with a Java back end. The Java back end does not use any frameworks such as Spring[Piv17c] but instead it uses various public libraries. The database layer is utilizing MySQL RDBMS.

During the last few years the cloud based electronic financial management and accounting market has been growing rapidly. Because of this Procountor has been growing rapidly and is currently the market leader in Finland and also has operations in Sweden, Norway and Denmark. The rapid growth means that the size of the development team has been growing in order to respond to the needs of the customers. Numerous recruitments and usage of external consultants have been made in Finland where most of the development currently resides. On top of that Procountor also utilizes development resources from Poland where external teams are working for Procountor.

Procountor software is a typical monolith application which consists of three layers. UI-layer, service or business logic layer, and database layer. On top of the main application that works in a browser there are also scheduled jobs, a public API and mobile applications. Scheduled jobs are long-running jobs that fetch bank data, create accounting based on the bank data and various other tasks. Public API enables third party developers to integrate with Procountor. Mobile applications allow users to login easier and do various little things such as invoice approval with their mobile phones.

The Procountor codebase carries technical debt from the previous years of development. Technical debt has accumulated from various sources. Most notably the first 10 years of development were done with limited resources and sometimes with

tight time schedule, which has resulted in legacy code. Even though continuous refactoring has been done during past few years the codebase still has today a lot of technical debt. The most problematic parts are old UI-components together with back end logic that has too big methods that do multiple things and have possible side effects.

On top of the technical debt, one problem is the lack of automated test coverage. Unit test coverage is not on a level that would be good enough. One of the reasons why unit test coverage is so low is that legacy parts of the codebase were created without thinking about testability. Methods are very long, and creating tests for these parts requires a lot of mocking. Efforts have been made to improve the unit test coverage and improvements continue to be made.

Integration testing and end-to-end testing were introduced to the codebase only a couple of years ago. Because of this the coverage coming from them both is still pretty small. Even though the coverage has been growing there is still a lot of work to be done in this area also. Especially end-to-end testing has been focused on and good results have been gained from both integration tests and end-to-end tests.

The lack of automated tests means that a lot of the testing is still done manually. Especially regression testing takes a lot of resources. This is something that has been recognized as a problem for a long time and because of that efforts have been made to get the automatic test coverage up.

Currently, releases to the Procountor main software are made about four times per year. This is mostly because of the lacking automated tests and the need for a long regression testing cycle. The release cycle has been noticed as too long and efforts are currently made to make releases at least once per month. However, a long term goal is to release even more often than monthly in order to stay competitive in the market and ensure the future growth. Microservice architecture is considered as one solution to this problem.

CI pipelines have been refined to achieve the goal of releasing more often. CI pipelines provide new build candidates automatically which then can be deployed manually to the testing servers. CI pipelines currently take a long time even though the automated test coverage is not optimal. On top of that, it is not always very clear who has broken the build if there are errors in the pipeline. The pipelines could be better optimized to improve the running time but the problem with who broke the build will stay.

Even if the problems with test coverage and pipelines are solved, releasing a monolithic system continuously is very challenging even though possible [Nai16]. A solution to this would be to transform the monolithic application to microservices architecture. The microservice architecture enables CD by having small independent services which in turn enables faster development [New15a]. Thus faster release cycles would be possible and the application would fill the needs of customers faster.

Even though the transition from monolith architecture to microservices architecture is challenging, the pros of microservices can be worth of the transformation. On top of the faster release cycles, microservices enable better scaling together with services that are easier to refactor, enable polyglot programming and are easier to understand. Scaling is important especially on the database level where currently sharding of the tables is being done to ensure better scalability in the future. Easier refactoring comes from the fact that the changes are contained inside one microservice thus making the refactoring changes limited. A single service is also easier to understand because it only has one responsibility which makes it clearer what is the purpose of the service.

Most of the development is done in Finland with teams that are co-located in the same office space which makes communication between these teams very easy. Procountor has also external teams working in Poland. This can add communicational problems between teams that are working in Finland and teams that are working in Poland. Microservices could ease the communication by having separated codebases based on the services. Polish teams could have their own set of microservices that they own and Finnish teams could have their own.

Future goals of the architecture are to enable fast development together with a scalable architecture that supports the growing number of users in different countries. Microservices can be seen as an enabler for both of these goals. This is why this case study is being done to show the challenges and possibilities that the transformation to microservices would provide.

5.2 Techniques used in proof of concept

In this section, the steps to decompose a business context from the existing monolith to a separated microservice is described. These steps can be repeated then for other business contexts in order to finally get rid of the monolith application. The steps were used when doing this proof of concept and it is recommended to follow these

steps when moving forward with the microservice architecture in order to minimize the regression effects that this transition might provide as a side-effect.

As described above, the first step is to identify the different bounded business contexts inside the application. It might be too hard and time consuming to identify all the different bounded contexts in the start of the transformation. Instead, the most obvious ones are be selected. These could be the bounded contexts that are going to be changed in the near future, the ones that could benefit from experimenting or the ones that are least tangled in the existing monolith. On top of these new functionalities should be implemented as microservices but in the context of this proof of concept only existing functionality is transformed to microservices.

For this proof of concept, two different bounded contexts were selected: invoices and attachments. The selection was done based on the previous criteria and also based on how to get most out of the proof of concept. Attachments were selected because the code around them is least tangled in the existing monolith making it an easy way to start the experimenting with microservices and thus gain more knowledge and expertise in the new architecture style. The transformation of attachments should highlight the good sides of microservices. Invoices were selected because even though they are tangled in the monolith with other business logic, they also give a good grasp of how to handle the harder transformations. A plan how to transform invoices to microservices will be presented later.

After the business contexts to be decomposed were selected the next step is to prepare the code that is to be transformed to a microservice by separating the business context from rest of the codebase and creating automated tests for it. If the business context is already well separated and modular inside the monolith, then this step can be skipped. However, as described before this is not the case always. In this proof of concept, the attachments were modular inside the monolith thus making the transformation relatively easy and fast.

The codebase for attachments was well tested with automatic unit and integration tests thus making the transition to microservice easy. Existing tests could be ported and moved to the new microservice acting as a way to ensure that the business rules were met even after the transformation. Existing integration tests were used as a template for the end-to-end tests in order to verify the whole microservice. As the business requirements stay same, it was easy to convert these tests to go through the REST API that the attachment microservice publishes.

When the codebase around a business context has been transformed to a microser-

vice, the next task is to handle communication between the monolith and the newly created microservice. At least in the beginning of the transformation, every microservice most likely has also to communicate with the monolith. There are two ways to handle this communication, building an anti-corruption layer or having a client between the monolith and the microservice without the anti-corruption layer.

When there is legacy code that does not enable easy communication between the monolith and microservice, then an anti-corruption layer between the monolith and microservice has to be made. This layer will transform the calls coming from legacy code to microservices so that the microservice does not get polluted with the bad design choices that were made previously. Vice versa the calls that the microservice makes to monolith can be transformed through the anti-corruption layer into the monolith without polluting the microservice.

In some cases also direct calls without the usage of the anti-corruption layer can be made between the monolith and microservice. This requires that both of them have similar structures and the differences in data models are not huge. REST APIs together with REST clients can be used in this case to handle the communication between microservice and monolith. If two way communication is required, then both the monolith and microservice should publish a REST API and have a REST client.

With invoices the anti-corruption layer is required because of the legacy code that resides in the monolith. However, as the attachment codebase was already in good shape in the monolith, there was no need to build an anti-corruption layer. Instead REST API and REST clients could be used. This means that if the data structures are already in good shape in the monolith, the transformation comes easier because there is no need for the anti-corruption layer which takes more time to build than just simple clients. However, developers should not shy away from building the anti-corruption layer when needed in order to ensure that the data structures inside the microservices are as good as possible.

As a part of the proof of concept also the transformation in the production environment was considered. Techniques such as canary deployments and feature flags can be used to ensure the validity of the newly developed microservice. Even with good automated test coverage, it is possible that the production environment exposes bugs that were introduced during the transition. With these two techniques the impact of the bugs can be minimized.

Canary deployment makes it possible to roll out the microservice first only to a small

set of users. During this time the microservice should be monitored extensively and possible error situations will only effect a part of the users thus minimizing the effects of bugs. If everything seems to be working correctly, a bigger number of users can be directed to microservice and eventually all of the traffic can go through the microservice.

Together with canary deployments feature flags can be used when making the final transition. The final transition should be handled with feature flags in the monolith. The functionality that was transitioned to microservice should be wrapped in a feature flag. This feature flag can be then used to disable the feature from monolith and instead use the newly developed microservice. This makes it easy to turn the feature back on in monolith in cases when the microservice is not behaving correctly. After the possible problems in microservice has been fixed the feature can be turned off from monolith and the code from monolith can be removed together with the feature flag. At this point, the transformation of the microservice can be considered done.

5.3 Proof of concept

The proof of concept consists of two business contexts attachments and invoices. Attachments were transitioned from the monolith to microservices. For invoices, a plan was made to how to handle the transition because this part of the application is so tangled that it is not possible in the scope of this proof of concept to finish the transformation. On top of the transformation and the plan how to transform tangled bounded contexts also communication between the microservices and monolith is addressed. The main focus is however on the transition and refactoring of the codebase around these two bounded contexts which are invoices and attachments.

As said, the focus of proof of concept is technical and organizational challenges are not in the scope of this. Procountor should make a clear plan how to handle the organizational challenges if the transition to microservices will be made. On top of the organizational challenges, running microservices in production environment provides its own challenges. These challenges are such as aggregated logging and monitoring. They will not be handled as part of this proof of concept because the focus is in the architectural challenges and not in the operational side.

5.3.1 Attachment microservice proof of concept

The transition work started from attachments as the code regarding attachments is already loosely coupled and has a good modular structure in the monolith. Also, the test coverage is on a good level thus making the transition easier. The choice to start from the easier one was made to give some experience about microservices and how the transition should be handled. This experience can then be used to solve more complex transformation cases.

The technology stack selected for attachments consist of Spring Boot[Piv17b], Kotlin[Jet17] and MySQL[Ora17b]. Spring Boot is a project that is part of the Spring framework[Piv17c]. Spring Boot uses the same components as the main Spring framework but it favors convention over configuration and makes it easy to start the development by providing an opinionated way of Spring framework [Piv17b]. When a project is built, it provides an embedded server which makes the running of the microservice easy thus making Spring Boot a good solution for microservices.

Kotlin was selected as the programming language. Kotlin is interoperable with Java and provides also multiple good improvements over Java such as the need for less boilerplate code, null-safety, smart casts, data classes, no raw types, proper function types and much more [Jet17]. Converting existing code from Java to Kotlin is easy because of the interoperability. Using Kotlin instead of Java also gives a nice example of the polyglotism provided by microservices.

MySQL was selected as the relational database management system. The choice was made because the author was already familiar with MySQL and even though microservices provide the possibility to use different database solutions MySQL has been good enough already in the monolithic application. Attachment data is saved to disk and MySQL only contains the metadata around attachments. If there were any scaling issues with MySQL and storing attachments, replacing MySQL would be easy because there are only a couple of database tables around attachments.

For data access code Jooq[Dat17] was used. Jooq generates code from the database thus making it easy to write SQL statement and at the same ensuring type safety. Jooq differs from the ORM (Object-relational mapping) tools in that way that the developer actually writes the SQL and has control over the SQL clauses instead of giving the control to the tool that can make inefficient queries. As attachments require dynamic queries based on the access rights, Jooq was a perfect choice for this microservice.

The architecture inside the attachment microservice follows a typical Spring based back end architecture. The microservice is divided to controller, service and data access layers. Controller and data access layers are thin and all of the business logic is in the service layer.

In the controller layer, the attachment microservice provides a simple REST API for the consumers of the microservice. The API provides CRUD (create, read, update, delete) functionalities together with a search functionality. Because of the simple API, the complexities are hidden from the clients that want to use the attachment microservice. The API also provides clear modular boundaries around the business context thus making it loosely coupled from the rest of the application codebase.

Listing 1: Attachment Controller example

```
@RestController
@RequestMapping("/attachments")
class AttachmentController(val service: AttachmentService) {

   @ApiOperation(value = "Get an attachment with id", response =
       Attachment::class)
   @GetMapping("/{attachmentId}")
    fun getAttachment(@PathVariable attachmentId: Int): Attachment {
        return service.getAttachment(attachmentId)
    }

   @ApiOperation(value = "Create a new attachment. Returns the created
       attachment", response = Attachment::class)
    @PostMapping
    fun saveAttachment(@RequestBody attachment: Attachment): Attachment {
        return service.saveAttachment(attachment)
    }
   // Rest of endpoints excluded from this example
}
```

The above code snippet, shows an example of a controller with Spring Boot and Kotlin as the programming language. The controller contains all the endpoints that the REST API for attachments publishes for other development teams to use. The following endpoints are excluded from the example above: deleting, searching and updating attachments. The annotations in the top of the class definition declare that an HTTP endpoint is exposed and where the endpoint is exposed. So all

the attachment microservice endpoints are exposed under the /attachments URL. Similarly, getting an attachment with id 123 from the microservice requires a call to /attachments/123 with the request method GET. Creating a new attachment requires a POST call to /attachments.

Documentation for the attachment microservice is done using Springfox [Spr17] which extends Swagger [Sma17]. Swagger is a popular tool to document REST APIs. ApiOperation annotations on top of the endpoint functions declare the documentation that will be appended to the Swagger UI which publishes to documentation. Good documentation is necessary for microservices in order to make them easy to use for the developers from other teams.

Listing 2: Spring Boot microservice main class

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableSwagger2
class AttachmentMicroserviceApplication

fun main(args: Array<String>) {
    SpringApplication.run(AttachmentMicroserviceApplication::class.java,
        *args)
}

@Bean
fun attachmentDocs(): Docket {
    return Docket (DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build()
            .pathMapping("/")
}
```

The above code snippet shows the main class for the Spring Boot attachment microservice. The microservice is started from the main function. SpringBootApplication annotation on top of the class declares that this is a Spring Boot Application. The annotation scans all the classes and makes them usable through annotations.

EnableDiscoveryClient annotation enables the attachment microservice to be discov-

ered by different discovery services. These services can be such as Eureka [Net17], Consul [Has17] or Zookeeper [The17]. A registry makes it possible for the clients to find the microservices and thus making it possible to change the place of the microservice or serve the microservice from multiple places.

EnableSwagger2 annotation together with the attachmentsDocs function publishes the documentation related to the attachment microservice. The function attachmentsDocs makes it possible for the developers to specify and add more documentation to the endpoints if needed. The documentation will be published as Swagger 2 documentation and it can be viewed through Swagger UI. Swagger UI provides an clean and simple HTTP UI for viewing the documentation.

In order to enable communication from the monolith to the newly created attachment microservice a new client that handles the communication has to be created. Because the existing codebase around attachments was already in a good shape, there is no need to build an anti-corruption layer but instead a simple client will do the job. A separate library to help the building of clients was made available as a separate module. The library contains the attachment data models. This library can then be used easily by the different clients that have to utilize the attachment microservice. The library containing the data models should be versioned and updated by the team who develops the attachment microservice.

Listing 3: Attachment Client example

```
public class AttachmentClient {

    private Client client;

    @Autowired
    private DiscoveryClient discoveryClient;

    public String serviceUrl() {
        InstanceInfo instance =
            discoveryClient.getNextServerFromEureka("attachments", false);
        return instance.getHomePageUrl();
    }
    // Actual calls extracted from this example. Will be shown later.
}
```

The above code snippet shows the attachment client which uses the aforementioned

attachment data library. In this example, the most interesting parts are the DiscoveryClient which uses Eureka to find the attachment microservice. Because the attachment microservice has the EnableDiscoveryClient annotation we can use the discovery client to find the microservice and utilize it as shown below.

Listing 4: Attachment Client calls example

```
public Attachment getAttachment(int id) {
    Response response = client.target(serviceUrl() + id)
            .request(MediaType.APPLICATION_JSON)
            .get();
    return response.readEntity(Attachment.class);
}

public Attachment saveAttachment(Attachment attachment) {
    Response response = client.target(serviceUrl())
            .request(MediaType.APPLICATION_JSON)
            .post(Entity.entity(attachment, MediaType.APPLICATION_JSON));
    return response.readEntity(Attachment.class);
}
```

With the usage of Eureka DiscoveryClient, which provides the URL to the attachment microservices, building a client that uses the attachment microservice is trivial. Listing 4 is a very simple example which excludes the error handling and possible usage of circuit breakers such as Hystrix that could and should be used in a real case. The attachment client utilizes Jersey [Ora17a] REST client to make the calls. Jersey makes it easy to do calls to REST APIs as can be seen from the above example. As we can see building REST clients is easy when utilizing the existing tools and frameworks that are available to developers.

5.3.2 Transformation plan for invoices

The second part of the proof of concept is how to transform tangled bounded contexts out from the monolith. This bounded context is invoices. Because of the size of the codebase and the time constraints for this research only a plan on how to handle these situations will be presented.

As the whole transformation is very long and it can be hard to justify financially this kind of transition it makes sense to split the transition into smaller pieces. Each

of these steps provides value to the organization by making the codebase easier to change. The transition can be separated to actions as shown in Figure 12. The rectangles with red background are phases of the transition in where the functionality still lives inside the monolith. The rectangles with the yellow background are after the transition to microservices has been made.

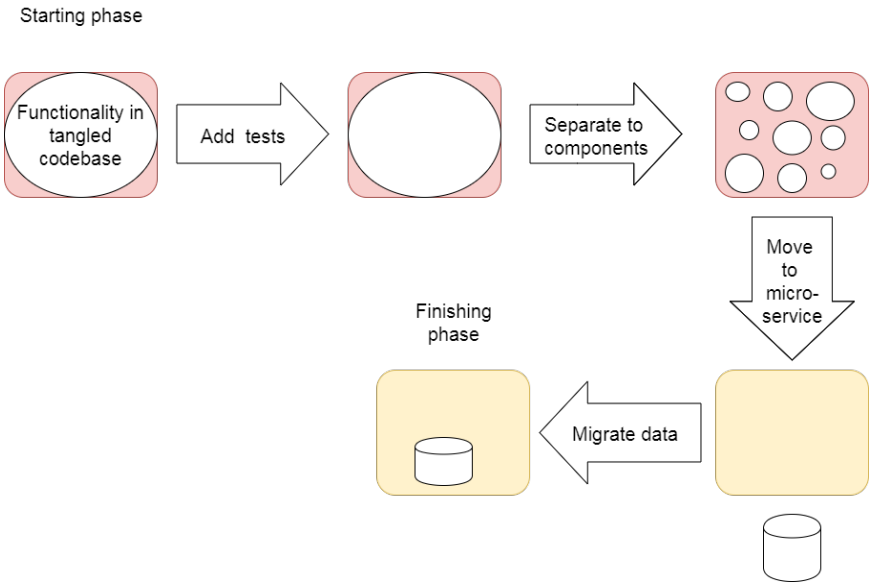

Figure 12: A plan how to transform parts of monolith to microservices

The first step is to add test coverage to the missing pieces. Like previously stated most of the focus should be on fast running unit tests to give fast feedback for developers and to ensure that the tests are not brittle. These unit tests coupled together with a few integration tests and end-to-end tests ensure that there are no regression issues during the transition.

After the testing coverage is at an accepted level, the next step is to separate the functionality into components inside the monolith. This means breaking down the possible god classes to smaller classes and utilizing the tools that programming languages give developers to create barriers between the different components in order to achieve high cohesion and loose coupling. Components should be focused on one business context and encapsulate their data inside the component.

Achieving this kind of architecture is possible for example, in Java by using visibility modifiers and packages. For example, in Procountor most of the Java classes are declared public. This means that they can be accessed from the whole module thus possibly resulting in low cohesion and tight coupling. Instead, classes could be

declared protected and with the usage of packages, they could then only be used inside that package. This gives clear interfaces to a business context and limits the possible introduction of bad coding habits.

The next action is to take the modular component and extract it from the monolith to a separate microservice. After the functionality is in a clearly modular component inside the monolith, transforming it to microservice should not be too challenging as we already saw with the attachments case. As Figure 12 on page 52 shows the database migration is not done yet at this phase. Instead, the microservice will still use the same database momentarily before the last phase. As explained before this is done in order to avoid multiple data migrations if the service boundaries require changes. However, this last phase should not last for too long and the data migration should be done as soon as possible.

The final transition which is the data migration enables the last good parts of microservices such as scalability, separate deployments and polyglot implementations. After this is done, canary deployments and feature flags can be used to deploy and test the microservice in production before fully transforming the traffic to the microservices. Also, as previously mentioned, clients and possibly anti-corruption layer between the monolith and the new microservice has to be built.

A plan about transforming invoices to microservice will be made. First, the role of invoices in Procountor is explained together with a small example how the invoice functionality should be separated to different microservices. After that, a closer look at the current situation is made and then solutions to the problematic parts of the architecture will be made.

Invoices have a very central part in financial management. Invoices are the basis of accounting and in Procountor accounting is updated automatically based on the invoices and the different accounting rules that have been. Thus, one can imagine a lot of business logic is revolving around the invoices and their different states.

Invoices can be separated into different types such as sales, purchase, travel, expense invoices and so on. These different types have a lot of similar qualities with each other even though there are also differences. Thus it does not make sense to separate the invoices based on their type to different microservices. Instead, the core functionality that is almost same for every invoice should be kept together in one place to avoid too much of duplication.

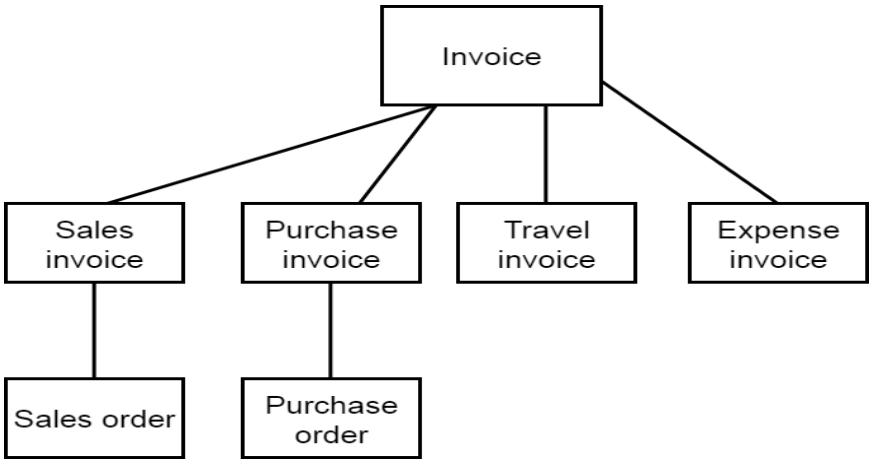

Figure 13: A subset of invoices in Procountor

Figure 13 displays a subset of different invoices that are in Procountor. These different types share common fields with each other and also have type specific fields. Validations and the creation of accounting also differs based on the invoice type.

The separation should be based on the factors that differentiate these invoices from each other. For example, purchase invoices can have an approval circulation which differs from the approval of a sales invoice or travel invoice. However, the data from the invoice that is needed for the business logic of approving is only a small subset of the all the data that an invoice has inside. On top of that, approval circulation requires knowledge about the next acceptor of the invoice, based on that the circulation service could alert the acceptor that she has to approve the invoice. After all the acceptors have accepted the invoice the circulation ends and the invoice status should be changed through the invoice service.

In the case when there is one service handling the purchase invoice circulation and another service handling the basic CRUD operations, considering invoices, it can seem that now these two services are tightly coupled as they both handle the invoice information. However, they can be made loosely coupled by having two different views on the invoice. The invoice circulation service does not need to know the same things about the invoice as the service that is used to create, read, update and delete invoices. Circulation service only needs to know the status of the invoice whether it is ready to be approved and the list of the approvers. This means that the two services are loosely coupled.

This work can be done inside the monolith without even going through the tran-

sition towards microservices. By using Java's visibility modifiers and packages it is possible to have separate packages for invoice CRUD operations and for the invoice circulation. This way it is possible to separate the concerns already inside the monolith. This requires restraint from the developers to not break the architecture because it is very easy for the developers to just change the visibility modifiers to public and thus breaking the architecture. Transitioning the code around business context to a separate microservice would make it a lot harder to break.

The current situation has multiple flaws starting from God Class [Cun13]. A God Class is a class that has too many responsibilities and It is a well known anti-pattern. These kinds of God Classes are problematic because they are hard to understand and modify. Another problem is that a lot of the logic is currently in the data access layer which can lead to confusions. On top of these problems, the test coverage is lacking leading to possible regression bugs when modifying existing code.

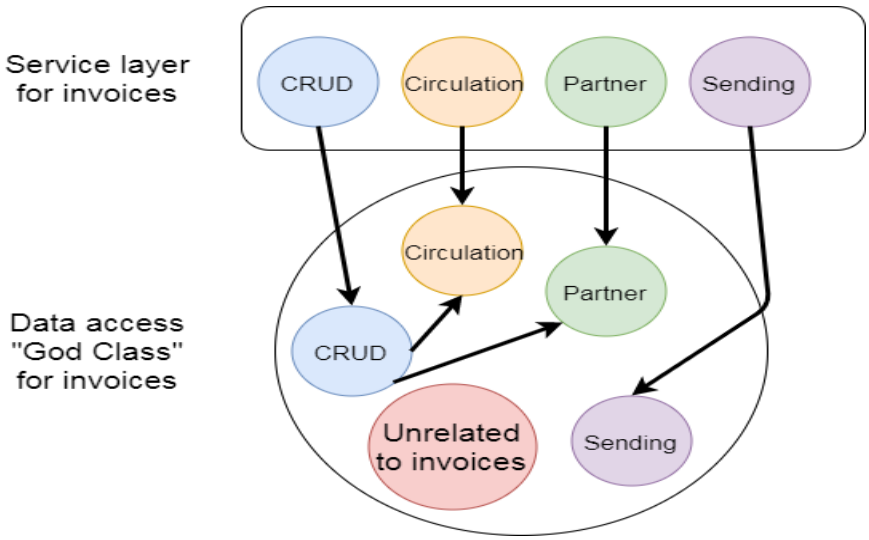

Figure 14: Current codebase relating to invoices.

Figure 14 illustrates the current codebase around invoices. Most of the functionality resides inside one God Class that has business logic together with data access logic. This makes the code very hard to understand and possible bugs can occur during the development. Two main objectives are to separate the business logic from the data access code and to split the God Class into smaller classes which have well defined responsibilities instead of cramming everything that has something to do with invoices to one class. There are even methods in the God Class which do not have much to do with invoices. The whole service layer is quite thin and acts only as a passage to the God Class which does all the work.

Testing methods that are too long and do multiple things is very hard. In order to make the code testable, these methods have to be split up and separated to smaller methods. Luckily Java IDEs provide good refactoring tools that aid this process. However, it is better to first write at least a couple of unit tests before refactoring functionality into smaller methods. After the functionality has been split up to smaller methods unit testing them is a lot easier.

When the unit tests are in place, it is time to move the functionality in to separate classes and packages using the Java visibility modifiers as explained before. Again the use of refactoring tools that the different Java IDEs provide is advised in order to minimize the manual errors that might happen during this period. The goal of the separation and visibility limitation is to make the codebase loosely coupled and cohesive.

After the visibility to the functionality is restricted with packages and there is a clear interface to the bounded context, it is possible to call the architecture as a modular monolith [Bro15]. Modular monolith is similar to component based architecture in the sense that it emphasizes the separation of concerns, high cohesion and low coupling [HC01]. In monolith everything is still running in the same process as opposed to microservices where every service is running in its own process. Microservices provide, then even more advantages such as easier upgrades, replacements and various other advantages together with challenges that were discussed before. Even though modular monolith does not provide all the advantages that microservices provide, it still has a place in the transition because it can be seen as a good stepping stone towards microservices. This was previously shown with attachments. They were already well separated which made the transitioning of the functionality from monolith to microservice easy. One should not try to take a too big bite at a time but instead, move in small steps in order to prevent regression bugs.

Figure 15 on page 57 shows the ideal architecture when the functionality for invoices is still inside the monolith. The God Class inside the data access layer has been removed and instead, there are packages for every separate concern and inside these packages only some of the classes are public and most of them are protected thus making them visible only inside the package. Business logic now lives in the service layer and services communicate with each other if they have to fetch data from another service instead of just communicating with each other in the data access layer. The example contains only a part of the invoice functionality.

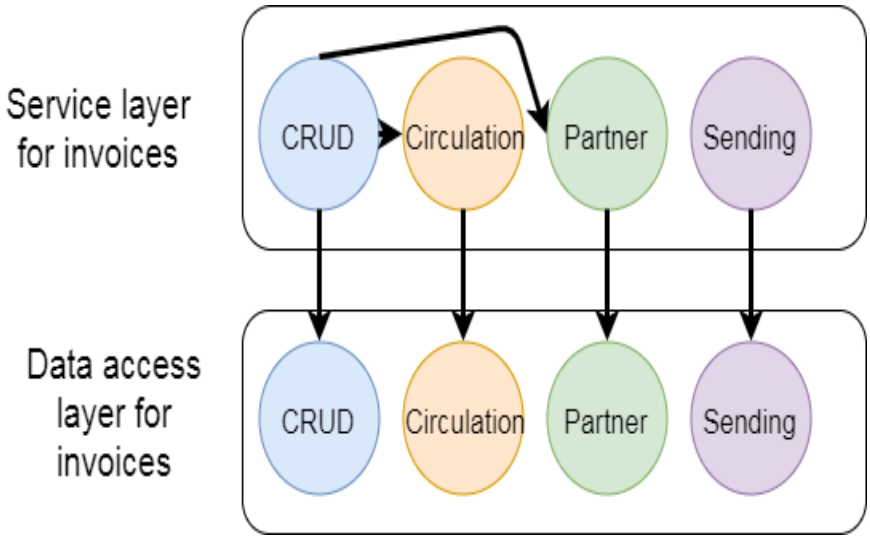

Figure 15: The ideal architecture for invoices inside the monolith.

Next step is the transformation of the invoice components to microservices. It is not necessary to move all the components straight to microservices. Moving them one by one reduces the risk of regression bugs and gives more experience about microservices. It makes sense to start from the components which are updated more frequently or could benefit from the other advantages of microservice architecture. In the case of invoices, the functions with most business logic are the CRUD and circulation services which could be a good starting point for the transformation to microservices.

After the transition to microservices, an anti-corruption layer between the monolith and the new microservices has to be built. This anti-corruption layer will transform the data structures that are inside the monolith to the newly refactored data structures that the microservices are handling. This way the legacy of the monolith is not passed on to the new microservices.

Figure 16 on page 58 shows the architecture after two of the components have been transformed to microservices. Most of the logic would still be in the monolith but slowly the whole functionality around invoices would be transformed to microservices. After everything considering invoices has been transformed to microservices the anti-corruption layer can be removed as communication considering invoices between monolith and microservices is not needed anymore.

Utilizing the plan outlaid in this section enables the transition towards microservices. The biggest challenges are moving from the legacy codebase towards components

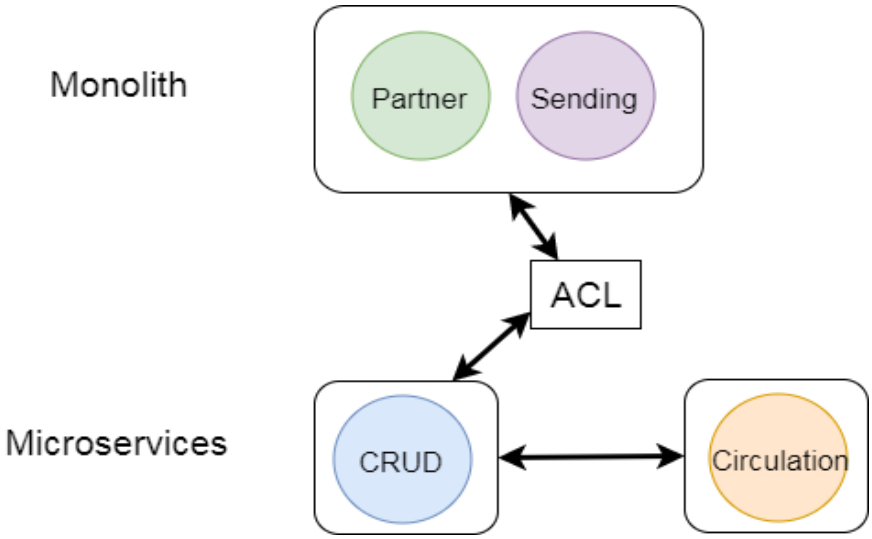

Figure 16: Invoice architecture after two transitions to microservices.

and the operational and organizational challenges that the microservice approach comes with. However, with time and resources, these challenges can be solved.

5.4 Results of proof of concept

In this section, the results from the proof of concept will be presented. The workload of the attachment microservice transition is discussed together with the challenges that the transition provided. Also, the biggest challenges that revolve around the plan that was made for the invoice transition will be presented.

The time spent doing the attachment microservice transition was measured. The amount of time spent was about three weeks of development time from one developer. This workload can only be used as a very rough estimate in the future as every functionality is different and they have a different kind of complexity. Some of the transitions might be faster, and some of them might take a lot longer time. Also, the time to polish and test the new microservice in a testing environment and in the production environment has to be taken into account.

On top of the time spent transforming one bounded context to microservice, there are also various other tasks that have to be completed in order to make a production ready transformation. Setting up Kubernetes or similar container orchestration is required to make sure that the microservice is running properly in production. Also building aggregated logging and monitoring takes some time. These tasks are however of such nature that once they are done, it is easy to replicate them to other

microservices so they are one-off tasks.

Together with the one-off tasks in order to get most out of microservices, building CI and CD pipelines are required for each microservices. This was not done for the attachment microservice. It can be estimated that building and polishing the pipelines is smaller task than the transformation of part of the codebase. These pipelines can also be copied for other microservices.

One thing that does not come clear from this kind of proof of concept is the complexity of running multiple microservices in the production environment. The challenges that the distributed computing comes with are not evident when there is just one microservice. This means that extra time has to be spent thinking about how to handle these challenges and how the microservice architecture shapes up when the number of microservices increases. For example, transactions can provide big challenges and there has to be a good plan how to handle them.

As we can see from the time estimates, transforming a functionality that is already in good shape inside the monolith to microservice is not a huge task. This does not, however, mean that the transition is not easy. The biggest challenges are not just moving the code to a microservice but operating it together with multiple other microservices and running this kind of setup in production.

Before the transitioning from monolith to microservices can be made it is better to first move towards a modular monolith. The road to modular monolith poses its problems. Moving from the legacy codebase to components inside the monolith has challenges because of the missing automated test coverage and the quality of the code inside the monolith. These kinds of big refactoring tasks take a lot of time. However, to support the development of new features, the refactoring is needed anyway whether or not the transition towards microservices is done. Based on previous experiences this kind of testing and refactoring effect can take from two weeks up to one month of development time from one team of developers. Depending of course on the size and the quality of the codebase.

Based on the proof of concept it was noticed that, when the bounded context is transitioned to microservice it will be easier for developers to understand as it is well separated. Also, the microservice can be deployed separately which gives the freedom to make faster releases. Faster releases are powered by CI and CD pipelines which are enabled by the microservice architecture. The amount of testing that has to be done before releasing a microservice to production is lower than with monolith thus setting up CD. Microservices also enable polyglot implementations, giving the

possibility to use the programming language that is best suited to the job instead of committing long term to a single programming language or framework.

The transformation plan that was presented for invoices was evaluated by development manager and a software architect from Procountor. The plan received a good evaluation, and it can be used if the transformation from monolith to microservices is made. It provides good steps towards the end goal by increasing the code quality already inside the monolith.

The reliability of the results depends on a lot on the fact that what is the scale that the results are evaluated on. There are two different scales. The first one is on the scale of one microservice. The second scale is whether the architecture is going to be better if the whole codebase is transitioned to microservices.

From transforming just one service, it is possible to say that the cohesion and loose coupling is more likely to stay that way when the bounded context is transformed to microservice. It is encouraging that the actual transformation of attachments went smoothly and there were no big challenges during it. From this point of view, it seems likely that microservices are the correct choice.

On the other hand, based on this proof of concept, it is very hard to say how the microservice architecture would shape for Procountor when there are numerous microservices. This is something that needs continuous planning and taking care of the architecture. Training, building libraries, and tools to support the developers building the microservices also take a lot of effort and have to be planned carefully to succeed with microservices.

5.5 Future Suggestions

Even though the proof of concept seems promising, the challenges that were described before should not be forgotten when making the actual decision whether to transform to microservices or not. The decision comes down to which architectural approach provides challenges which are easier to solve. Some of the biggest software companies in the world have made the decision that the challenges that microservices present are easier to solve than the ones with the monolith. As the microservice architecture pattern is still young and there are not enough studies about their usage for multiple years, it is hard to say whether the transition is good in the long run. Also, the size of these companies such as Amazon and Netflix is so big that they cannot be compared to Procountor. It is good to note that also

smaller companies have made the transition to microservices even though the most vocal ones are of course the biggest companies which lead the way.

Microservices could solve the challenges regarding the slow release cycle and innovation. They could also ensure that the cohesion and loose coupling stays in the codebase. Because of these reasons, it makes sense to aim for the microservice architecture in the long run. The transition is going to take a lot of time and resources because of the size of the codebase and the legacy parts of the codebase require cleaning up before or during the transformation.

One challenge with Procountor codebase and the transition to microservices is that the current situation in the monolith is that most of the functionalities are not ready for this transition yet. There are parts that are loosely coupled but still some of the codebase is tightly coupled with each other, the automated test coverage is not high enough, and there are other anti-patterns in the code such as God Classes. These parts of the legacy codebase have to be first refactored into components inside the monolith before the transition can begin.

Even though there are challenges in the codebase, it makes sense to start the transition towards microservices. If the bounded context that is loosely coupled is transformed to a microservice, it will stay loosely coupled also in the future. If these parts that are currently loosely coupled after refactoring are left in the monolith, it is possible that they decay over time. Developers have to have a lot of self-discipline to ensure that the good quality remains in the codebase. In the monolith, the ownership of the codebase is not always clear which makes it very likely that the self-discipline is not just there. This can lead to situations where quick fixes are implemented in a way that the good code qualities are forgotten.

The way to move forward is to identify the parts of the codebase that have good quality already. These parts should be easy to move to microservices. The proof of concept made for the attachment microservice is probably the easiest starting point. Building the aggregated logging, monitoring, CI and CD pipelines together with setting up orchestration tools is something that should be done when setting up the attachment microservice to be production ready. Running this one microservice in production gives a lot of experience about microservices and can show potential problems with the setup.

After the possible problems with attachment microservice are solved, the next step should be to pick up another bounded context that is easily extractable. By moving slowly this way the confidence towards running microservices in production should

increase together with the tooling getting better.

At the same time that the microservice transition is happening efforts should be made to make the existing monolith towards a modular monolith. Also, automated test coverage should be brought up. Both of these actions support the transition to microservices, but they also are useful in the monolith because they improve the code quality and make it easier to develop new features.

Together with the technical challenges that are being solved, Procountor should also pay attention to the organizational challenges that the transition to microservices provides. The human perspective is important together with the new technical skills that are required from the teams.

Teams should own the microservice that they have built. This means that the mindset has to change drastically. The change of mindset might scare some employees, but the change is only for the better because while it requires more responsibility it also gives more freedom to the teams. Teams can now self-organize and select their technologies and take care of the microservice in production.

Transition to microservices requires new technical skills from the teams. Even though some of the tools that are useful when building, such as Docker, are already in use at Procountor, the adaptation of these tools has to continue and spread. This means that everyone should feel comfortable using these tools. Many new tools are also introduced such as Kubernetes. These new tools bring their challenges and time has to be devoted to get the needed knowledge about these tools.

The organizational changes can help with the recruitment process of new developers. When there are more technologies that can be used in the codebase, the range of developers which can be recruited becomes bigger. Developers are typically interested in working with new technologies and having the possibility to try out new tools. The introduction of new technologies is easier in microservices. This could help Procountor to continue to attract good developers in the future.

To conclude, the future suggestions is that the transition to microservices should be considered strongly. The transition period will be long, and it provides many challenges but by starting out slowly and ironing out all the problems in the beginning while still thinking about the big picture, this transition can provide a competitive advantage in the future for Procountor. The competitive advantage comes from the possibility to innovate and release changes faster and easier.

6 Conclusions

In this thesis I studied the transformation from monolith to microservices. Monolithic architecture faces multiple problems when the size of the codebase grows large. The development starts slowly getting harder and the large codebase intimidates new developers as it takes a long time to get familiar with the monolith. Trying to use continuous delivery is hard with monolithic architecture thus leading to slower release cycles. This together with slower development means that the companies using monolithic applications are slower to react to the changing markets and the needs of the customers.

In the last few years, a possible solution to this problem has emerged. Microservice architecture takes a new view to building large scale applications. The codebase is split up into hundreds of small services which each have a single responsibility and the services are built around business contexts thus making it easier for new and old developers to know what each part of the application does. Each of these services can be developed, deployed and scaled independently from other services.

The small size of the services enables innovation. Companies utilizing microservices can try out new things and get feedback from the users faster as they have continuous deployment pipelines together with monitoring tools that give business data from the microservices which can be utilized to make decisions on whether to continue the development of certain features.

Even if the innovation possibilities of microservices is a big advantage it comes with a price. Thus microservices are not a free lunch. Microservice architecture provides multiple technical and organizational challenges that have to be solved in order to get the benefits out of the architectural style.

The question whether to transition from monolith to microservices comes down to whether the challenges that the microservice architecture provide are easier to solve for the organization than the challenges that the monolithic architecture provides.

In order to find out whether microservices are suitable for Procountor a proof of concept was made. The proof of concept consisted of transforming one existing bounded context to microservices and designing a plan how to transform bigger and more tangled contexts from monolith to microservices.

The transformation in chapter 4 was made on attachments which were already in a good shape inside the monolith. The transformation process was pretty straightforward and did not introduce any new challenges than the ones that were listed before.

The resulting attachment microservice ensures that the separation of concerns, loose coupling and cohesion will stay with attachments.

The second part of the proof of concept was to make a plan how to transform tangled business contexts out from the monolith to microservices. This proof of concept was made on invoices. Invoices are a central part of Procountor's software and the codebase around them have a lot of legacy code and data structures are sub-optimal.

The plan could be split to four actions. The first action is to add automated tests around the business context to limit the number of regression bugs. The second action is to separate the existing codebase inside monolith to components. These components are still inside the monolith and they will be transformed to microservices in the next action which is the move to microservices.

In the third step, the components which are inside monolith are transformed to microservices one by one. The final action is to move the data from monolith to microservice. As data migrations are very expensive it makes sense to do the data migration after the microservice has already been functioning for a while.

The results from the proof of concept were encouraging. The actual transformation of attachments to microservices went smoothly. The resulting attachment microservice will maintain the good code qualities such as separation of concerns, loose coupling and cohesion because of the hard boundaries that microservices impose. As we could see from the proof of concept the polyglot approach is also possible.

Building fast CI and CD pipelines for attachments is now possible which means that changes to attachments could be deployed to production a lot faster. This solves the problem around slow release cycles and makes innovation possible.

The main problems are still the high coupling and low cohesion in parts of the codebase together with low test coverage. Working on both of these problems while at the same time moving towards a modular monolith and slowly moving these components inside the modular monolith to microservices seems like the best way to go. The plan that was made for invoices can be used for other bounded contexts also. The transformation towards microservices will be a long journey but the long time should not be seen as a roadblock. It is better to move slowly and gain the needed experience and skills that are required for the transformation as it is a challenge for the whole organization. This transformation can give the competitive edge over other companies as the automated financial management market continues to grow.

References

AF09 Abbott, M. L. and Fisher, M. T., *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise.* Pearson Education, 2009.

Bal00 Baldwin, C. Y., C. K. B., *Design rules: Volume 1, The power of modularity.* MIT Press cop, 2000.

BHJ16 Balalaie, A., Heydarnoori, A. and Jamshidi, P., Microservices architecture enables devops: migration to a cloud-native architecture. *IEEE Software*, 33,3(2016), pages 42–52.

Bri96 Briand, L. C., M. S. B. V. R., Property-based software engineering measurement. *IEEE Transactions on Software Engineering, 22(1)*, 1996, pages 68–86.

Bro15 Brown, Simon, Modular Monolith, 2015. `http://www.codingthearchitecture.com/presentations/sa2015-modular-monoliths`. [5.7.2017]

Cal14 Calcado, P., Building Products at SoundCloud Part I: Dealing with the Monolith, 2014. `https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith`. [22.4.2017]

CJS16 Carneiro Jr, C. and Schmelmer, T., Polyglot services. In *Microservices From Day One*, Springer, 2016, pages 177–184.

Cle14 Clemson, T., Testing Strategies in a Microservice Architecture, 2014. `http://martinfowler.com/articles/microservice-testing/`. [11.4.2017]

Coh10 Cohn, M., *Succeeding with agile: software development using Scrum.* Pearson Education, 2010.

Con68 Conway, M. E., How do committees invent. *Datamation*, 14,4(1968), pages 28–31.

Cun13 Cunningham & Cunningham, Inc., God Class, 2013. `http://wiki.c2.com/?GodClass`. [3.7.2017]

Dat17 Data Geekery GmbH, Jooq, 2017. https://www.jooq.org/. [5.7.2017]

Doc17 Docker Inc, Docker, 2017. https://www.docker.com/. [2017-05-10]

Doe17 Doerrfeld, Bill, How To Control User Identity Within Microservices, 2017. http://nordicapis.com/how-to-control-user-identity-within-microservices/. [5.7.2017]

Ela17a Elastic, Kibana, 2017. https://www.elastic.co/products/kibana. [18.4.2017]

Ela17b Elastic, Logstash, 2017. https://www.elastic.co/products/logstash. [18.4.2017]

Eva04 Evans, E., *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

FB99 Fowler, M. and Beck, K., *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

Fea04 Feathers, M., *Working effectively with legacy code*. Prentice Hall Professional, 2004.

Fow02 Fowler, M., *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc, 2002.

Fow04 Fowler, M., Strangler Application, 2004. https://www.martinfowler.com/bliki/StranglerApplication.html. [25.4.2017]

Fow14a Fowler, M., Circuit Breaker, 2014. https://martinfowler.com/bliki/CircuitBreaker.html. [20.4.2017]

Fow14b Fowler, M., Microservice Prerequisites, 2014. https://martinfowler.com/bliki/MicroservicePrerequisites.html. [6.5.2017]

Fow15a Fowler, M., Microservice Premium, 2015. https://martinfowler.com/bliki/MicroservicePremium.html. [7.4.2017]

Fow15b Fowler, M., Monolith First, 2015. https://martinfowler.com/bliki/MonolithFirst.html. [7.4.2017]

Fow15c Fowler, M., Microservice trade offs, 2015. https://martinfowler.com/articles/microservice-trade-offs.html. [7.5.2017]

Has17 HashiCorp, HashiCorp Consul, 2017. https://github.com/hashicorp/consul. [10.5.2017]

Hay08 Hayes, B., Cloud computing. *Communications of the ACM*, 51,7(2008), pages 9–11.

HC01 Heineman, G. T. and Councill, W. T., Component-based software engineering. *Putting the pieces together, Addison-Westley*, page 5.

HL95 Hürsch, W. L. and Lopes, C. V., Separation of concerns.

Hun00 Hunt, A., *The pragmatic programmer*. Pearson Education India, 2000.

Ihd15 Ihde, S., From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture, 2015. https://www.infoq.com/presentations/linkedin-microservices-urn. [26.3.2017]

Jet17 JetBrains s.r.o, Kotlin, 2017. https://kotlinlang.org/. [17.6.2017]

JNS16 Jaramillo, D., Nguyen, D. V. and Smart, R., Leveraging microservices architecture by using docker technology. *SoutheastCon, 2016*. IEEE, 2016, pages 1–5.

Kha15 Kharenko, A., Monolithic vs Microservices Architecture, 2015. https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59. [2017-05-06]

KJP15 Krylovskiy, A., Jahn, M. and Patti, E., Designing a smart city internet of things platform with microservice architecture. *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 2015, pages 25–30.

LF14 Lewis, J. and Fowler, M., Microservices a definition of this new term, 2014. https://martinfowler.com/articles/microservices.html. [26.3.2017]

Mar03 Martin, R. C., *Agile software development: principles, patterns, and practices*. Pearson Education, 2003.

Mar09a Martin, R. C., *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

Mar09b Martin, R. C., The Single Responsibility Principle, 2009. http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle. [28.3.2017]

Mar09c Martin, R. C., *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

Mau15a Mauro, T., Adopting Microservices at Netflix: Lessons for Architectural Design, 2015. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/. [26.3.2017]

Mau15b Mauro, T., Adopting Microservices at Netflix: Lessons for Team and Process Design, 2015. https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/ [18.3.2017]

Mer14 Merkel, D., Docker: lightweight linux containers for consistent development and deployment. *Linux Journal, 2014(239), 2*, 2014.

Mun15a Munns, C., DevOps at Amazon: Microservices, 2 Pizza Teams, and 50 Million Deploys a Year, 2015. https://www.slideshare.net/TriNimbus/chris-munns-devops-amazon-microservices-2-pizza-teams-50-million-deplo [28.3.2017]

Mun15b Munns, C., Microservices at Amazon, 2015. https://www.slideshare.net/apigee/i-love-apis-2015-microservices-at-amazon-54487258. [26.3.2017]

MW16 Montesi, F. and Weber, J., Circuit breakers, discovery, and api gateways in microservices. *arXiv preprint arXiv:1609.05830*.

Nai16 Naik, V., Architecting for Continuous Delivery, 2016. https://www.thoughtworks.com/insights/blog/architecting-continuous-delivery. [2017-05-07]

Net15 Netflix, Inc., Netflix at Github, 2015. https://github.com/Netflix. [28.3.2017]

Net16 Netflix Inc., Hystrix, 2016. https://github.com/Netflix/hystrix. [26.3.2017]

Net17 Netflix Inc, Netflix Eureka, 2017. https://github.com/Netflix/eureka. [2017-05-10]

New14 Newman, S., Demystifying Conway's Law, 2014. https://www.thoughtworks.com/insights/blog/demystifying-conways-law. [6.4.2017]

New15a Newman, S., *Building microservices*. O'Reilly Media, Inc., United States of America, 2015.

New15b Newman, S., Microservices For Greenfield?, 2015. http://samnewman.io/blog/2015/04/07/microservices-for-greenfield/. [7.4.2017]

Nor03 Nordberg, M. E., Managing code ownership. *IEEE software, 20(2)*, 2003, pages 26–33.

Nov17 Novak, A., Going to Market Faster: Most Companies Are Deploying Code Weekly, Daily, or Hourly, 2017. https://blog.newrelic.com/2016/02/04/data-culture-survey-results-faster-deployment/. [7.4.2017]

NSS14 Namiot, D. and Sneps-Sneppe, M., On micro-services architecture. *International Journal of Open Information Technologies*, 2,9(2014).

OAB12 Olsson, H. H., Alahyari, H. and Bosch, J., Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE, 2012, pages 392–399.

Ora17a Oracle Corporation, Jersey, 2017. https://jersey.github.io/. [20.6.2017]

Ora17b Oracle Corporation, MySQL, 2017. https://www.mysql.com/. [12.5.2017]

Pap03 Papazoglou, M. P., Service-oriented computing: Concepts, characteristics and directions. *Web Information Systems Engineering, WISE 2003. Proceedings of the Fourth International Conference on IEEE.*, 2003, pages 3–12.

Piv17a Pivotal Software Inc., RabbitMQ, 2017. `https://www.rabbitmq.com/`. [2017-04-25]

Piv17b Pivotal Software, Inc., Spring Boot, 2017. `https://projects.spring.io/spring-boot/`. [17.6.2017]

Piv17c Pivotal Software, Inc., Spring Framework, 2017. `https://projects.spring.io/spring-framework/`. [13.4.2017]

Pro17 Procountor Oy, Procountor, 2017. `http://www.procountor.com/`. [12.5.2017]

Ric15a Richardson, C., Microservices, 2015. `http://microservices.io`. [26.3.2017]

Ric15b Richardson, C., Microservices, database per service, 2015. `http://microservices.io/patterns/data/database-per-service.html`. [11.4.2017]

Ric15c Richardson, C., Microservices decompose by business capability, 2015. `http://microservices.io/patterns/decomposition/decompose-by-business-capability.html`. [26.3.2017]

Ric15d Richardson, C., Microservices, shared database, 2015. `http://microservices.io/patterns/data/shared-database.html`. [11.4.2017]

Ric15e Richardson, C., Monolithic architecture, 2015. `http://microservices.io/patterns/monolithic.html`. [26.3.2017]

Ric16 Richards, M., *Microservices Antipatterns and Pitfalls.* O'Reilly Media, Inc., 2016.

Ric17 Richardson, C., Microservices, Database per service, 2017. `http://microservices.io/articles/scalecube.html`. [13.4.2017]

RMMB15 Ranchal, R., Mohindra, A., Manweiler, J. G. and Bhargava, B., Radical strategies for engineering web-scale cloud solutions. *IEEE Cloud Computing*, 2,5(2015), pages 20–29.

Sch14 Schauenberg, D., Development, Deployment and Collaboration at Etsy, 2014. `https://speakerdeck.com/mrtazz/development-deployment-and-collaboration-at-etsy`. [2017-05-09]

Sma17 SmartBear Software, Swagger, 2017. `http://swagger.io/`. [18.6.2017]

SMD16 Stubbs, J., Moreira, W. and Dooley, R., Distributed systems of microservices using Docker and Serfnode. *In Science Gateways (IWSG), 2015 7th International Workshop on IEEE*, 2016, pages 34–39.

Spr17 Springfox, Springfox, 2017. `http://springfox.github.io/springfox/`. [18.6.2017]

Sti15 Stine, M., Migrating to cloud-native application architectures, 2015.

TBB+15 Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F. and Edmonds, A., An architecture for self-managing microservices. *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. ACM, 2015, pages 19–24.

The17 The Apache Software Foundation, Apache Zookeeper, 2017. `https://zookeeper.apache.org/`. [10.5.2017]

Til15 Tilkov, S., Don't start with a monolith, 2015. `https://www.martinfowler.com/articles/dont-start-monolith.html`. [7.4.2017]

Vaa17 Vaadin Ltd., Vaadin, 2017. `https://vaadin.com`. [12.5.2017]

VGC+15 Villamizar, M., Garces, O., Castro, H., Salamanca, L., Casallas, R. and Gil, S., Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *Proc. Computing Colombian Conference*, 2015, pages 583–590.

WC10 Wampler, D. and Clark, T., Guest editors' introduction: multiparadigm programming. *IEEE Software*, 27,5(2010), pages 20–24.

Wol16 Wolff, E., *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016.

Woo14 Wootton, B., Microservices - not a free lunch!, 2014. `http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html`. [2017-05-06]